Formale Sprachen und Komplexität (FSK)

und

Theoretische Informatik für Studierende der Medieninformatik (TIMI)

Sommersemester 2024

PD Dr. David Sabel

Überarbeitet von
Prof. Dr. Jasmin Blanchette
Jannis Limperg
Luca Maio

Lehr- und Forschungseinheit für
Theoretische Informatik und Theorembeweisen
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67
80538 München

Stand: 9. September 2024

Inhaltsverzeichnis

1. Einleitung **1**

2. Grundlagen **2**
2.1. Natürliche Zahlen, Alphabete, Wörter und Sprachen . . . 2
2.2. Relationen . . . 6
2.3. Funktionen . . . 8
2.4. Abzählbarkeit und Überabzählbarkeit . . . 8
2.5. Asymptotische Notation . . . 10

I. Formale Sprachen und Automatentheorie **11**

3. Grammatiken und die Chomsky-Hierarchie **12**
3.1. Grammatiken . . . 13
3.2. Die Chomsky-Hierarchie . . . 16
3.2.1. Erzeugung des leeren Worts und ε-Produktionen . . . 17
3.2.2. Beziehungen zwischen den Typ i-Sprachen . . . 19
3.3. Das Wortproblem . . . 20
3.4. Weitere Probleme für formale Sprachen . . . 22
3.5. Syntaxbäume . . . 22
3.6. Die Backus-Naur-Form für Grammatiken . . . 24

4. Reguläre Sprachen **25**
4.1. Deterministische endliche Automaten . . . 25
4.2. DFAs akzeptieren reguläre Sprachen . . . 28
4.3. Nichtdeterministische endliche Automaten . . . 29
4.4. Reguläre Sprachen können durch NFAs erkannt werden . . . 30
4.5. Überführung von NFAs in DFAs . . . 31
4.6. NFAs mit ε-Übergängen . . . 34
4.7. Reguläre Ausdrücke . . . 37
4.8. Zusammenfassende Darstellung der Formalismen für reguläre Sprachen . . . 42
4.9. Das Pumping-Lemma . . . 42
4.10. ⋆ Der Satz von Myhill und Nerode . . . 46
4.11. Minimierung von Automaten . . . 48
4.12. Abschlusseigenschaften der regulären Sprachen . . . 52
4.13. Entscheidbarkeitsresultate zu regulären Sprachen . . . 53

5. Kontextfreie Sprachen **54**
5.1. ⋆ Einfache Operationen auf CFGs . . . 54

5.2. Chomsky-Normalform . . . 56
5.2.1. ⋆ Entfernen von Einheitsproduktionen . . . 56
5.2.2. ⋆ Herstellen der Chomsky-Normalform . . . 58
5.3. ⋆ Greibach-Normalform . . . 61
5.3.1. Herstellen der Greibach-Normalform . . . 61
5.4. ⋆ Widerlegen der Kontextfreiheit . . . 64
5.5. ⋆ Abschlusseigenschaften kontextfreier Sprachen . . . 70
5.6. Effiziente Lösung des Wortproblems: Der CYK-Algorithmus . . . 70
5.7. Kellerautomaten . . . 73
5.7.1. Akzeptanz durch Endzustände . . . 76
5.7.2. Äquivalenz von Kellerautomaten und kontextfreien Sprachen . . . 78
5.7.2.1. ⋆ Kontextfreie Sprachen werden durch Kellerautomaten erkannt . . . 78
5.7.2.2. Kellerautomaten, die maximal 2 Symbole auf den Keller legen 79
5.7.2.3. ⋆ Kellerautomaten akzeptieren kontextfreie Sprachen . . . 79
5.7.2.4. Kellerautomaten und kontextfreie Sprachen . . . 81
5.8. Deterministisch kontextfreie Sprachen . . . 82
5.9. ⋆ Entscheidbarkeitsresultate . . . 84
5.9.1. Ein entscheidbares Problem . . . 85

6. Kontextsensitive und Typ 0-Sprachen 86
6.1. ⋆ Die Kuroda-Normalform für kontextsensitive Grammatiken . . . 86
6.2. Turingmaschinen . . . 87
6.3. Linear beschränkte Turingmaschinen und kontextsensitive Sprachen . . . 91
6.3.1. ⋆ Beweis des Satzes von Kuroda . . . 92
6.4. Turingmaschinen und Typ 0-Grammatiken . . . 94
6.5. ⋆ LBA-Probleme . . . 95

7. Zusammenfassung und Überblick 97

II. Berechenbarkeitstheorie 99

8. Der intuitive Berechenbarkeitsbegriff 100

9. Turings Modell der Berechenbarkeit 103
9.1. Turingmaschinen und Turingberechenbarkeit . . . 103
9.2. Mehrspuren- und Mehrband-Turingmaschinen . . . 104
9.2.1. Mehrspuren-Turingmaschinen . . . 104
9.2.2. Mehrband-Turingmaschinen . . . 106
9.3. ⋆ Modulare Konstruktion von Turingmaschinen . . . 108
9.3.1. Notation . . . 108
9.3.2. Einfache Rechenoperationen . . . 108
9.3.3. Komposition von Turingmaschinen . . . 109

10. ⋆ LOOP-, WHILE- und GOTO-Berechenbarkeit 112
10.1. LOOP-Programme . . . 112
10.1.1. Syntax von LOOP-Programmen . . . 112
10.1.2. Semantik von LOOP-Programmen . . . 112
10.1.3. LOOP-Berechenbarkeit . . . 114
10.1.4. Eigenschaften von LOOP-Programmen und der LOOP-Berechenbarkeit . . . 114
10.2. WHILE-Programme . . . 116
10.2.1. Syntax der WHILE-Programme . . . 116
10.2.2. Semantik der WHILE-Programme . . . 116
10.2.3. WHILE-Berechenbarkeit . . . 116
10.3. GOTO-Programme . . . 117
10.3.1. Syntax der GOTO-Programme . . . 117
10.3.2. Semantik der GOTO-Programme . . . 118
10.3.3. GOTO-Berechenbarkeit . . . 119
10.4. Äquivalenz der WHILE- und GOTO-Berechenbarkeit . . . 119
10.5. Simulation von Turingmaschinen mit GOTO-Programmen . . . 120
10.5.1. Darstellung von Wörtern als natürliche Zahlen . . . 120
10.5.2. Darstellung von TM-Konfigurationen mit GOTO-Programmvariablen 121

11. ⋆ Primitiv und μ-rekursive Funktionen 125
11.1. Primitiv rekursive Funktionen . . . 125
11.1.1. Definition der primitiv rekursiven Funktionen . . . 125
11.1.2. Einfache Beispiele und Konstruktionen primitiv rekursiver Funktionen . . . 125
11.1.3. Primitiv rekursive Funktionen zur Gödelisierung von Tupeln natürlicher Zahlen . . . 126
11.1.4. Äquivalenz von LOOP-berechenbaren und primitiv rekursiven Funktionen . . . 129
11.2. μ-Rekursive Funktionen . . . 130

12. ⋆ Schnell wachsende Funktionen: Die Ackermannfunktion 133

13. Unentscheidbarkeit 138
13.1. Gödelisierung von Turingmaschinen . . . 140
13.2. Das Halteproblem . . . 140
13.2.1. Unentscheidbarkeit des speziellen Halteproblems . . . 140
13.2.2. Reduktionen . . . 142
13.2.3. Unentscheidbarkeit des Halteproblems . . . 143
13.2.4. Halteproblem bei leerer Eingabe . . . 143
13.3. Der Satz von Rice . . . 143
13.4. Das Postsche Korrespondenzproblem . . . 144
13.4.1. MPCP $\leq$ PCP . . . 145
13.4.2. $H \leq$ MPCP . . . 146
13.4.3. Varianten und Bemerkungen zum PCP . . . 147
13.5. Universelle Turingmaschinen . . . 148
13.6. ⋆ Unentscheidbarkeitsresultate für Grammatik-Probleme . . . 148

III. Komplexitätstheorie **152**

14. Einleitung, Zeitkomplexität, $\mathcal{P}$ und $\mathcal{NP}$ **153**
14.1. Deterministische Zeitkomplexität . . . 153
14.1.1. Uniformes vs. logarithmisches Kostenmaß . . . 155
14.2. Nichtdeterminismus . . . 156
14.3. Das P-vs.-NP-Problem . . . 156

15. NP-Vollständigkeit **158**
15.1. Definition der NP-Vollständigkeit . . . 158
15.1.1. Polynomialzeit-Reduktionen . . . 158
15.2. Satz von Cook . . . 160

16. Eine Auswahl NP-vollständiger Probleme **165**
16.1. Das 3-CNF-SAT-Problem . . . 165
16.2. Das CLIQUE-Problem . . . 167
16.3. Das INDEPENDENT-SET-Problem . . . 169
16.4. Das VERTEX-COVER-Problem . . . 171
16.5. ⋆ Das SET-COVER-Problem . . . 172
16.6. ⋆ Das SUBSET-SUM-Problem . . . 173
16.7. ⋆ Das KNAPSACK-Problem . . . 175
16.8. ⋆ Das PARTITION-Problem . . . 175
16.9. ⋆ Das BIN-PACKING-Problem . . . 176
16.10. Das DIRECTED-HAMILTON-CYCLE-Problem . . . 177
16.11. Das UNDIRECTED-HAMILTON-CYCLE-Problem . . . 180
16.12. Das TRAVELING-SALESPERSON-Problem . . . 181
16.13. Das GRAPH-COLORING-Problem . . . 181

Literatur **184**

1. Einleitung

Dieses Skript fasst die wesentlichen Inhalte der Vorlesungen „Formale Sprachen und Komplexität“ und „Theoretische Informatik für Studierende der Medieninformatik“ zusammen, die im Sommersemester 2023 an der Ludwig-Maximilians-Universität München gehalten werden.

Da die Vorlesung „Theoretische Informatik für Studierende der Medieninformatik“ nur einen Teil der Vorlesung umfasst, sind die Kapitel und Abschnitte oder auch einzelne Sätze bzw. deren Beweise, die für Studierende der Medieninformatikinnen und Medieninformatiker nicht prüfungsrelevant sind, durch einen roten Stern $\star$ gekennzeichnet.

Die Primärliteratur für die Vorlesung ist Uwe Schönings Buch *Theoretische Informatik – kurz gefasst* (Sch08). Das Skript orientiert sich an dessen Inhalt und strukturellen Aufbau.

Die Vorlesung behandelt drei wesentliche Teilgebiete der Theoretischen Informatik. Teil I befasst sich mit Formalen Sprachen und Grundlagen der Automatentheorie. Teil II befasst sich mit der Frage, welche Probleme *überhaupt* mit dem Rechner *gelöst* werden können (die Berechenbarkeitstheorie). Teil III befasst sich mit der Frage, welche Probleme *effizient lösbar* sind und was selbiges heißt (die Komplexitätstheorie).

Ziel der Veranstaltung ist es eine Einführung in und einen Überblick über die drei Themengebiete zu geben.

Vor den einzelnen Teilen ist das Kapitel 2 zu finden. Hier fassen wir wesentliche mathematische Grundlagen zusammen, die im Grunde bekannt sein dürften. In der Vorlesung werden nicht alle dieser Grundlagen wiederholt, sondern zum Teil dem Selbststudium überlassen.

2. Grundlagen

In diesem Kapitel werden wesentliche Grundlagen zusammenfassend dargestellt und Notationen festgelegt. Die Inhalte sind im Wesentlichen auch in (Sch08, Anhang: Mathematische Grundlagen) nachzulesen.

2.1. Natürliche Zahlen, Alphabete, Wörter und Sprachen

Wir bezeichnen die Menge der natürlichen Zahlen einschließlich der Null mit $\mathbb{N}$, d.h. $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ und mit $\mathbb{N}_{>0}$ bezeichnen wir die Menge der natürlichen Zahlen ohne Null, d.h. $\mathbb{N}_{>0} = \{1, 2, 3, \ldots\}$.
Um Aussagen für alle natürlichen Zahlen zu beweisen, verwenden wir oft das Prinzip der vollständigen Induktion:

Definition 2.1.1 (Beweisprinzip der Vollständigen Induktion)**.** *Um zu zeigen, dass eine Aussage $A(n)$ für jede natürliche Zahl $n \in \mathbb{N}$ gilt, genügt es, die folgenden beiden Aussagen zu zeigen:*

1. *(Induktionsanfang): $A(0)$ gilt.*
2. *(Induktionsschritt): Für eine beliebige Zahl $n \in \mathbb{N}_{>0}$ gilt: Wenn für alle $m \in \mathbb{N}$ $A(m)$ gilt, dann gilt auch $A(n)$.*

Wir demonstrieren die vollständige Induktion:

Beispiel 2.1.2. *Für alle $n \in \mathbb{N}$ 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 zu zeigende Aussage $A(0)$ ist $\sum_{i=1}^{0} i = 0$. Dies folgt direkt aus der Definition der Summe.*
- *Induktionsschritt: Sei $n \in \mathbb{N}_{>0}$ eine beliebige, positive natürliche Zahl. Wir dürfen für alle $m < n$ $A(m)$ annehmen und müssen $A(n)$ zeigen. Da n positiv ist, existiert ein $k \in \mathbb{N}$ mit $n = k + 1$, und da $k < n$ ist, dürfen wir insbesondere $A(k)$ annehmen, also $\sum_{i=1}^{k} i = \frac{(k+1)k}{2}$. Wir müssen $A(k+1)$ zeigen, also $\sum_{i=1}^{k+1} i = \frac{(k+2)(k+1)}{2}$. Es gilt $\sum_{i=1}^{k+1} i = \left(\sum_{i=1}^{k} i\right) + k + 1$ nach Definition. Nach Annahme ist diese Summe gleich $\frac{(k+1)k}{2} + k + 1$. Durch Ausrechnen erhält man $\frac{(k+1)k}{2} + k + 1 = \frac{(k+1)k + 2k + 2}{2} = \frac{(k+2)(k+1)}{2}$, also insgesamt die zu zeigende Eigenschaft.*

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

Definition 2.1.3 (Alphabet, Wort, Konkatenation)**.** *Ein* Alphabet *ist eine endliche nicht leere Menge von* Zeichen *(oder* Symbolen*). Wir bezeichnen Alphabete oft mit dem Symbol Σ.*

Sei ein Alphabet Σ gegeben. Ein Wort *w über Σ ist eine endliche Folge von Zeichen aus Σ. Die leere Folge ist auch ein solches Wort und wird mit ε 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 Σ. Mit $\#_a(w) \in \mathbb{N}$ 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 Σ^* bezeichnen wir die Menge aller Wörter über Σ. Eine formale (rekursive) Definition für Σ^* ist die folgende, wobei Σ^i die Menge der Wörter der Länge i über dem Alphabet Σ bezeichnet. Die Definition der Mengen Σ^i ist:*

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

Schließlich definiere $\Sigma^* := \bigcup_{i \in \mathbb{N}} \Sigma^i$ und $\Sigma^+ := \bigcup_{i \in \mathbb{N}_{>0}} \Sigma^i$. Es gilt $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

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

Beispiel 2.1.4. *Sei $\Sigma = \{a, b\}$. Dann ist $\Sigma^0 = \{\varepsilon\}$, $\Sigma^1 = \Sigma$, $\Sigma^2 = \{aa, ab, ba, bb\}$ und $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \ldots\}$ und z.B. $aabbb \in \Sigma^*$ aber $abc \notin \Sigma^*$. Für $u = aab$ und $v = aabbba$ gilt $|u| = 3$, $|v| = 6$ und $u \cdot v = uv = aabaabbba$, sowie z.B. $\#_b(u) = 1$ und $\#_a(uv) = \#_a(u) + \#_a(v) = 2 + 3 = 5$.*

Satz 2.1.5. *Die Struktur $(\Sigma^*, \cdot)$ ist ein Monoid, d.h. eine Halbgruppe mit neutralem Element.*

Beweis. Die Eigenschaften einer Halbgruppe sind

1. Abgeschlossenheit bezüglich $\cdot$ (d.h. $u, v \in \Sigma^* \implies u \cdot v \in \Sigma^*$) und
2. Assoziativität von $\cdot$ (d.h. $(u \cdot v) \cdot w = u \cdot (v \cdot w)$).

Für ein neutrales Element e muss gelten $u \cdot e = e \cdot u = u$.

Für (1) sei $u \in \Sigma^i$ und $v \in \Sigma^j$, dann ist $u \cdot v \in \Sigma^{i+j}$ und daher in $u \cdot v \in \Sigma^*$. Für (2) sei $u = a_1 \cdots a_i$, $v = b_1 \cdots b_j$ und $w = c_1 \cdots c_k$ mit $i, j, k \in \mathbb{N}$ und $a_l, b_l, c_l \in \Sigma$. Dann gilt $(u \cdot v) \cdot w = a_1 \cdots a_i b_1 \cdots b_j c_1 \cdots c_k = u \cdot (v \cdot w)$. Schließlich gilt $u \cdot \varepsilon = \varepsilon \cdot u = u$, d.h. ε ist ein neutrales Element. □

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

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

Definition 2.1.6. *Sei Σ ein Alphabet und w ein Wort über Σ. Mit $\overline{w}$ bezeichnen wir das rückwärtsgelesene Wort w, d.h. $\overline{\varepsilon} = \varepsilon$, $\overline{a} = a$ für $a \in \Sigma$ und $\overline{aw} = \overline{w}a$ für alle $a \in \Sigma, w \in \Sigma^*$.*

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

Beispiel 2.1.7. *Sei $w =$ informatik ein Wort über $\Sigma = \{a, \ldots, z\}$. Dann ist $\overline{w} =$ kitamrofni und w ist kein Palindrom. Hingeghen ist $w' =$ reliefpfeiler ein Palindrom, denn $\overline{w'} =$ reliefpfeiler.*

Beispiel 2.1.8. *Für alle Wörter w und Symbole a gilt $\overline{wa} = a\overline{w}$. Dies lässt sich mit Induktion über $|w|$ zeigen: Wenn $|w| = 0$, so gilt $w = \varepsilon$ und damit $\overline{wa} = \overline{a} = a = a\overline{w}$. Für $|w| > 0$ existiert ein Wort v und Symbol b sodass $w = bv$. Dann gilt $\overline{bva} = \overline{va}b$ per Definition, $\overline{va}b = a\overline{v}b$ per Induktionshypothese und schließlich $a\overline{v}b = a\overline{bv} = a\overline{w}$ per Definition.*

Übungsaufgabe 2.1.9. *Zeige mit Induktion über $|w|$: Für alle Wörter w gilt $\overline{\overline{w}} = w$.*

Definition 2.1.10 (Sprache und Operationen auf Sprachen)**.** *Eine (formale)* Sprache *über dem Alphabet Σ ist eine Teilmenge von Σ^*. Oft bezeichnen wir Sprachen mit dem Buchstaben L (für „language"). Für Sprachen L, L_1 und L_2 über dem Alphabet Σ 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)*
- $\overline{L} := \Sigma^* \setminus L$ *(das* Komplement *zu L)*
- $L_1 \cdot L_2 = L_1L_2 := \{uv \mid u \in L_1 \text{ und } v \in L_2\}$ *(das* Produkt *von L_1 und L_2)*

Bemerkung 2.1.11. *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_1L_2 keine Paare, sondern konkateniert die Wörter w_1 und w_2 direkt zu einem neuen Wort.

Beispiel 2.1.12. *Sei $\Sigma = \{a, b\}$, $L_1 = \{a^i \mid i \in \mathbb{N}\}$ und $L_2 = \{b^i \mid i \in \mathbb{N}\}$. 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; $L_1L_2 = \{a^ib^j \mid i, j \in \mathbb{N}\}$, $L_2L_1 = \{b^ia^j \mid i, j \in \mathbb{N}\}$ und $L_1L_1 = L_1$.*
Für $L_1 = \{\spadesuit, \clubsuit, \diamondsuit, \heartsuit\}$ und $L_2 = \{7, 8, 9, 10, J, D, K, A\}$ stellt L_1L_2 eine Repräsentation der Spielkarten eines Skatblatts dar.

Definition 2.1.13. *Seien u, v Wörter über einem Alphabet Σ. Man sagt*

- *u ist ein* Präfix *von v, wenn es ein Wort w über Σ gibt, sodass $uw = v$.*
- *u ist ein* Suffix *von v, wenn es ein Wort w über Σ gibt, sodass $wu = v$.*
- *u ist ein* Teilwort *von v, wenn es Wörter w_1, w_2 über Σ gibt, sodass $w_1uw_2 = v$.*

Beispiel 2.1.14. *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 2.1.15 (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 $(L_1 \cup L_2) \in \mathcal{L}$,*

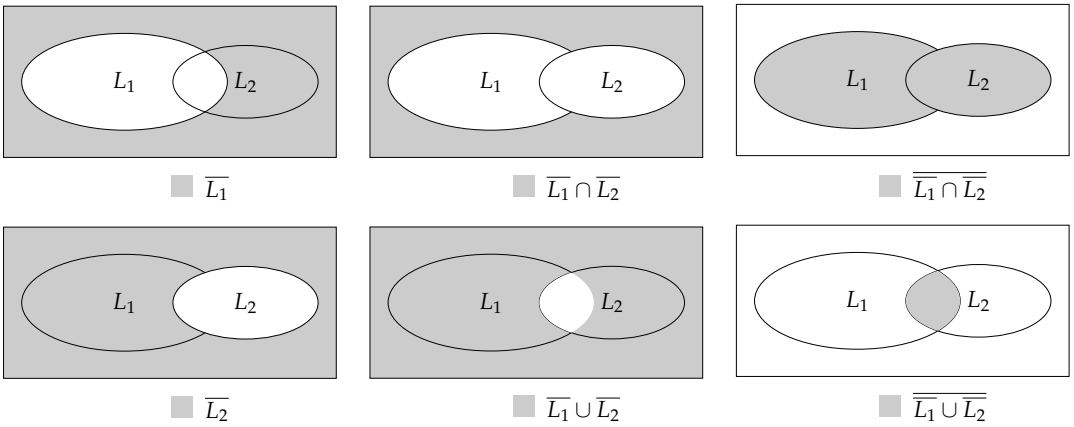
- Schnittbildung *g.d.w. aus* $L_1 \in \mathcal{L}$ *und* $L_2 \in \mathcal{L}$ *folgt stets* $(L_1 \cap L_2) \in \mathcal{L}$,
- Komplementbildung *g.d.w. aus* $L \in \mathcal{L}$ *folgt stets* $\overline{L} \in \mathcal{L}$ *und*
- Produktbildung *g.d.w. aus* $L_1 \in \mathcal{L}$ *und* $L_2 \in \mathcal{L}$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

Satz 2.1.16. *Sei die Klasse von Sprachen $\mathcal{L}$ abgeschlossen bezüglich Komplementbildung. Dann ist $\mathcal{L}$ abgeschlossen bezüglich Schnittbildung g.d.w. $\mathcal{L}$ abgeschlossen bezüglich Vereinigung ist.*

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

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

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:

$$L^0 := \{\varepsilon\} \qquad\qquad L^* := \bigcup_{i \in \mathbb{N}} L^i$$
$$L^i := L \cdot L^{i-1} \text{ für } i > 0 \qquad\qquad L^+ := \bigcup_{i \in \mathbb{N}_{>0}} L^i$$

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

Beispiel 2.1.17. *Sei $L = \{ab, ac\}$. Dann ist $L^2 = \{abab, abac, acab, acac\}$ (alle Wörter, die sich aus 2 Wörtern aus L bilden lassen) und $L^* = \{\varepsilon\} \cup \{ax_1ax_2 \cdots ax_i \mid i \in \mathbb{N}_{>0}, x_j \in \{b, c\}, j = 1, \ldots, i\}$.*

Beispiel 2.1.18. *Die Sprache*

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

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

2.2. Relationen

Seien R, R_1, R_2 binäre Relationen auf Σ^* (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

$$\begin{array}{lcl} R^0 & := & \{(w, w) \mid w \in \Sigma^*\} \text{ (die identische Abbildung)} \\ R^i & := & RR^{i-1} \text{ für } i > 0 \\ R^* & := & \bigcup\limits_{n \in \mathbb{N}} R^i \\ R^+ & := & \bigcup\limits_{n \in \mathbb{N}_{>0}} R^i \end{array}$$

Es gilt uR^*w g.d.w. $u = w$ oder es gibt $v_1, \ldots, v_n \in \Sigma^*$ und $n > 0$ mit $uRv_1, v_1Rv_2, \ldots, v_nRw$.

Definition 2.2.1. *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 Σ^* in (endlich oder unendlich viele) Äquivalenzklassen* disjunkt *zerlegt, d.h. $\Sigma^* = [w_1]_R \dot\cup [w_2]_R \dot\cup [w_3]_R \dot\cup \ldots$, wobei $\dot\cup$ die disjunkte Vereinigung bezeichne[1]. Der* Index *einer Äquivalenzrelation R (geschrieben als $Index(R)$) ist die Anzahl der verschiedenen Äquivalenzklassen, die R hat. Hierbei ist $Index(R) \in \mathbb{N} \cup \{\infty\}$. Wenn $Index(R) \neq \infty$, dann sagt man R hat einen* endlichen Index.

Beispiel 2.2.2. *Sei $\Sigma = \{a, b\}$ und* echtesPraefix *eine binäre Relation auf Σ^*, definiert durch: Für alle $u, v \in \Sigma^*$: u* echtesPraefix *v g.d.w. u ist ein Präfix von v und $|u| < |v|$.*

Die Relation echtesPraefix *ist transitiv: Seien $u, v, w \in \Sigma^*$ mit u* echtesPraefix *v und v* echtesPraefix *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* echtesPraefix *w gezeigt.*

Die Relation echtesPraefix *ist nicht reflexiv, da z.B. $\neg(\varepsilon$* echtesPraefix *$\varepsilon)$.*

Die Relation echtesPraefix *ist nicht symmetrisch, da z.B. aa* echtesPraefix *aab, aber $\neg(aab$* echtesPraefix *$aa)$.*

Die Relation echtesPraefix* *ist reflexiv, da w* echtesPraefix^0 *w für alle $w \in \Sigma^*$. Die Relation* echtesPraefix* *ist transitiv: Sei u* echtesPraefix^i *v und v* echtesPraefix^j *w, dann gilt u* echtesPraefix^{i+j} *w und daher u* echtesPraefix* *w. Die Relation* echtesPraefix* *ist nicht symmetrisch, da $\neg(a$* echtesPraefix^i *$ab)$ für alle $i \in \mathbb{N}$. Es gilt u* echtesPraefix$^*v$ *g.d.w. u ein Präfix von v ist.*

[1] d.h. man schreibt $A \dot\cup B$ g.d.w. $A \dot\cup B = A \cup B$ und $A \cap B = \emptyset$.

Beispiel 2.2.3. *Sei* $\Sigma = \{a, b\}$ *und* u gleicherAnfang v *g.d.w. u und v beginnen mit dem gleichen Buchstaben oder sind beide das leere Wort.*

gleicherAnfang *ist eine Äquivalenzrelation:*

- gleicherAnfang *ist reflexiv: Wir prüfen* u gleicherAnfang u *für alle* $u \in \Sigma^*$*: Wenn u mit a anfängt (d.h.* $u = av$ *für ein* $v \in \Sigma^*$*) gilt* u gleicherAnfang u*. Wenn u mit b anfängt (d.h.* $u = bv$ *für ein* $v \in \Sigma^*$*) gilt* u gleicherAnfang u*. Auch für* $u = \varepsilon$ *gilt* u gleicherAnfang u*.*
- gleicherAnfang *ist symmetrisch: Offensichtlich folgt aus* u gleicherAnfang v *auch* v gleicherAnfang u *für alle* $u, v \in \Sigma^*$*:*
- gleicherAnfang *ist transitiv: Sei* u gleicherAnfang v *und* v gleicherAnfang w *für* $u, v, w \in \Sigma^*$*. Dann fangen entweder* u, v, w *alle mit dem selben Buchstaben an (und* u gleicherAnfang w *gilt) oder* $u = v = w = \varepsilon$ *und auch dann gilt* u gleicherAnfang w*.*

Der Index von gleicherAnfang *ist* $Index(\text{gleicherAnfang}) = 3$*, denn es gibt drei disjunkte Äquivalenzklassen:*

- $[a]_{\text{gleicherAnfang}} = \{w \in \Sigma^* \mid a \text{ gleicherAnfang } w\} = \{au \mid a \in \Sigma^*\}$
- $[b]_{\text{gleicherAnfang}} = \{w \in \Sigma^* \mid b \text{ gleicherAnfang } w\} = \{bu \mid b \in \Sigma^*\}$
- $[\varepsilon]_{\text{gleicherAnfang}} = \{w \in \Sigma^* \mid \varepsilon \text{ gleicherAnfang } w\} = \{\varepsilon\}$

Lemma 2.2.4. *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 $i, j \in \mathbb{N}$ mit uR^iv und vR^jw. 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 $i \in \mathbb{N}$. 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 $i \in \mathbb{N}$ gilt. Sei $uR^{i+1}v$. Dann gibt es ein w mit uRw und wR^iv. 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 $R^{i+1} \subseteq R'$. □

Definition 2.2.5. *Seien R und S Äquivalenzrelationen auf* Σ^**. Dann ist R eine* Verfeinerung *von S, wenn für alle* $u, v \in \Sigma^*$ *gilt:* $uRv \implies uSv$ *(d.h.* $R \subseteq S$*).*

Satz 2.2.6. *Sei R eine Verfeinerung von S. Dann gilt* $Index(R) \geq Index(S)$*.*

Beweis. Sei $\Sigma^* = [u_1]_R \cup [u_2]_R \cup \ldots$ 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. $\Sigma^* = [u_1]_S \cup [u_2]_S \cup \ldots$ und damit gilt insbesondere, dass S nicht mehr disjunkte Äquivalenzklassen hat als R. □

Beispiel 2.2.7. *Sei* $\Sigma = \{a, b\}$*. Die Relationen* gleicherAnfang *und* gleicherAnfangUndEnde *seien definiert durch*

- u gleicherAnfang v *g.d.w. u und v beginnen mit dem gleichen Buchstaben oder sind beide das leere Wort.*

- u 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 2.2.3 gezeigt, der Nachweis für* gleicherAnfangUndEnde *funktioniert analog). Die Relation* gleicherAnfangUndEnde *ist eine Verfeinerung von* gleicherAnfang, *denn aus* u gleicherAnfangUndEnde v *folgt stets* u gleicherAnfang v. *In Beispiel 2.2.3 haben wir gezeigt, dass* $Index(\text{gleicherAnfang}) = 3$ *gilt. Für* gleicherAnfangUndEnde *gilt* $Index(\text{gleicherAnfangUndEnde}) = 5$, *denn* gleicherAnfangUndEnde *hat folgende disjunkten Äquivalenzklassen:*

- $[\varepsilon]_{\text{gleicherAnfangUndEnde}} = \{\varepsilon\}$
- $[a]_{\text{gleicherAnfangUndEnde}} = \{u \in \Sigma^* \mid u \text{ beginnt mit } a \text{ und endet mit } a\}$
- $[b]_{\text{gleicherAnfangUndEnde}} = \{u \in \Sigma^* \mid u \text{ beginnt mit } b \text{ und endet mit } b\}$
- $[ab]_{\text{gleicherAnfangUndEnde}} = \{u \in \Sigma^* \mid u \text{ beginnt mit } a \text{ und endet mit } b\}$
- $[ba]_{\text{gleicherAnfangUndEnde}} = \{u \in \Sigma^* \mid u \text{ beginnt mit } b \text{ und endet mit } a\}$

2.3. Funktionen

Seien D und Z Mengen. Eine Funktion $f : D \to Z$ 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 2.3.1. *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* $f : D \to Z$. *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.*

2.4. Abzählbarkeit und Überabzählbarkeit

Seien M_1, M_2 Mengen. Wir nennen M_1 und M_2 *gleichmächtig*, wenn es eine bijektive Abbildung $f : M_1 \to M_2$ gibt. Falls eine Menge M gleichmächtig wie $\mathbb{N}$ ist, dann heißt M *abzählbar unendlich*. Eine Menge M heißt *abzählbar*, wenn sie endlich oder abzählbar unendlich ist.

Lemma 2.4.1. *Eine Menge M ist genau dann abzählbar, wenn* $M = \emptyset$ *gilt oder es eine surjektive Abbildung* $f : \mathbb{N} \to M$ *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 $f : \mathbb{N} \to M$ und die Aussage gilt. Wenn M endlich ist, dann sei $M = \{a_0, \ldots, a_n\}$. Definiere $f : \mathbb{N} \to M$ 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 $x \in \mathbb{N}$ und f damit surjektiv.

Für die umgekehrte Richtung sei $f : \mathbb{N} \to M$ eine surjektive Abbildung. Wenn M endlich ist, dann ist M abzählbar. Wenn M nicht endlich ist, dann müssen wir zeigen, dass M gleichmächtig zu $\mathbb{N}$ ist. Dazu sei $g : \mathbb{N} \to \mathbb{N}$ rekursiv definiert durch $g(0) = 0$ und $g(n+1) = k$, wobei $k \in \mathbb{N}$ minimal gewählt ist, sodass $f(k) = f(g(n+1)) \notin \{f(g(0)), \ldots, f(g(n))\}$. Eine solche Zahl k gibt es, da M sonst endlich wäre. Die Konstruktion von g sichert zu, dass für alle $i \in \mathbb{N}$ gilt: $g(i+1) > g(i)$.
Wir zeigen, dass $h : \mathbb{N} \to M$ 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 $n \in \mathbb{N}$ daher minimal gewählt, sodass $f(n) = m$. Da $g(i+1) > g(i)$ für alle $i \in \mathbb{N}$, gibt es nur endlich viele Zahlen $i \in \{0, 1, \ldots, j\}$ mit $g(j) < n$. Daher gilt $g(j+1) = k \geq n$, wobei k minimal gewählt ist, sodass $f(k) \notin \{f(g(0)), \ldots, f(g(j))\}$. 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 2.4.2. *Seien M_1, M_2 abzählbar, dann ist auch $M_1 \times M_2$ abzählbar.*

Beweis. Wir zeigen zunächst, dass $(\mathbb{N} \times \mathbb{N})$ abzählbar ist. Die folgende Tabelle deutet Cantors 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 $h : \mathbb{N} \to (\mathbb{N} \times \mathbb{N})$ gibt[2]. Nun zeigen wir das Lemma: Da M_1 und M_2 abzählbar sind, gibt es surjektive Funktionen $f_i : \mathbb{N} \to M_i$ für $i = 1, 2$. Sei $g : (\mathbb{N} \times \mathbb{N}) \to (M_1 \times M_2)$ definiert durch $g(i, j) = (f_1(i), f_2(j))$. Schließlich definiere $g' : \mathbb{N} \to (M_1 \times M_2)$ als $g'(i) = g(h(i))$. Da g, h surjektiv sind, ist auch g' surjektiv. □

Satz 2.4.3. *Zu einem Alphabet Σ ist Σ^* 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 $|\Sigma| = s$) □

[2] Eine andere surjektive Funktion ist $h : \mathbb{N} \to (\mathbb{N} \times \mathbb{N})$ 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 $i, j \in \mathbb{N}$) zerlegen lässt.

Definition 2.4.4 (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 2.4.5. *Sei Σ ein Alphabet. Dann ist die Menge aller Sprachen über Σ gerade $\mathcal{P}(\Sigma^*)$.*

Satz 2.4.6. *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 $f : \mathbb{N} \to M$ und $g : \mathbb{N} \to \mathcal{P}(M)$. Definiere die Diagonalmenge $D := \{f(i) \mid f(i) \notin g(i), i \in \mathbb{N}\}$. 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 $n \in \mathbb{N}$ 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. □

2.5. Asymptotische Notation

Funktionen $f : \mathbb{N} \to \mathbb{N}$, welche die Komplexität von Algorithmen beschreiben, werden meist mithilfe der asymptotischen Notation angegeben. Wir betrachten hier nur die O-Notation (gesprochen „Groß-O-Notation“). Diese vernachlässigt konstante Faktoren und gibt eine asymptotische Abschätzung nach oben an. Genauer definiert $O(g)$ eine Klasse von Funktionen:

Definition 2.5.1. *Für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ 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 2.5.2. *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 $n \geq 1$).*

Teil I.

Formale Sprachen und Automatentheorie

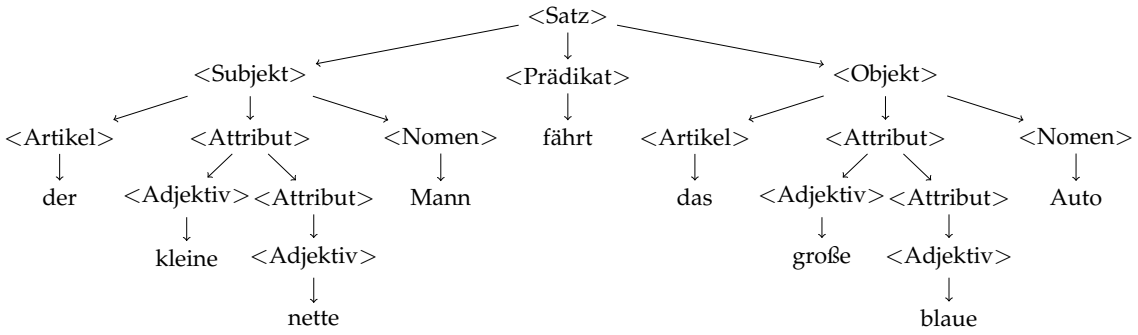
3. Grammatiken und die Chomsky-Hierarchie

Sei Σ ein Alphabet. Eine Sprache über Σ ist eine beliebige Teilmenge von Σ^*. Z.B. könnten wir für $\Sigma = \{(,), +, -, *, /, a\}$ die Sprache L_{ArEx} definieren, welche als Wörter die korrekt geklammerten arithmetischen Ausdrücke über Σ enthalte. 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 gewissen (sehr kleinen) 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>	→ ε	<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.

3.1. Grammatiken

Grammatiken sind endliche Mengen von Regeln der Form

„linke Seite“ $\to$ „rechte Seite“.

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

Grammatiken werden benutzt, um aus einer ausgezeichneten Variablen (dem sogenannten *Startsymbol*) ein Wort über Σ *abzuleiten*. Ableiten meint hierbei mehrfaches (endliches) 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 Σ, die vom Startsymbol aus abgeleitet werden können, definiert die von der Grammatik erzeugte Sprache.

Definition 3.1.1 (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 *oder* Nichtterminalsymbole*).*
- *Σ ist ein Alphabet von Zeichen (auch* Terminale *oder* 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 $\ell \to r$ 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 $\ell \to r$ als Paare (ℓ, 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*. Seien u, v Satzformen. Wir definieren $u \Rightarrow_G v$ (u* geht unter Grammatik G unmittelbar in v über*), falls es Satzformen w_1, w_2 gibt, sodass*

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

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

Beispiel 3.1.2. *Sei* $G = (\{E, M, Z\}, \{+, *, 1, 2, (,)\}, P, E)$ *mit*

$$\begin{aligned} P = \{ & E \to M, \\ & E \to E + M, \\ & M \to Z, \\ & M \to M * Z, \\ & Z \to 1, \\ & Z \to 2, \\ & Z \to (E)\} \end{aligned}$$

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:*

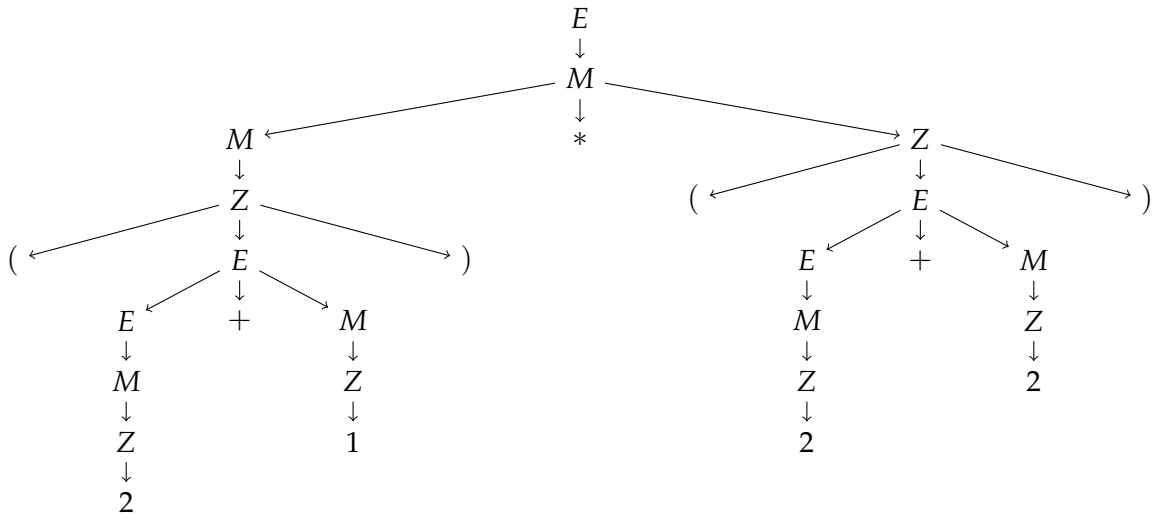
$$\begin{aligned} & E \Rightarrow M \Rightarrow M*Z \Rightarrow Z*Z \Rightarrow Z*(E) \Rightarrow Z*(E+M) \Rightarrow (E)*(E+M) \Rightarrow (E)*(E+Z) \Rightarrow \\ & (E+M)*(E+Z) \Rightarrow (M+M)*(E+Z) \Rightarrow (M+M)*(M+Z) \Rightarrow (M+M)*(Z+Z) \Rightarrow \\ & (M+M)*(Z+2) \Rightarrow (M+Z)*(Z+2) \Rightarrow (M+Z)*(2+2) \Rightarrow (Z+Z)*(2+2) \Rightarrow \\ & (2+Z)*(2+2) \Rightarrow (2+1)*(2+2) \end{aligned}$$

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:*

$$\begin{aligned} & E \Rightarrow M \Rightarrow M*Z \Rightarrow Z*Z \Rightarrow (E)*Z \Rightarrow (E+M)*Z \Rightarrow (M+M)*Z \Rightarrow (Z+M)*Z \Rightarrow \\ & (2+M)*Z \Rightarrow (2+Z)*Z \Rightarrow (2+1)*Z \Rightarrow (2+1)*(E) \Rightarrow (2+1)*(E+M) \Rightarrow \\ & (2+1)*(M+M) \Rightarrow (2+1)*(Z+M) \Rightarrow (2+1)*(2+M) \Rightarrow (2+1)*(2+Z) \Rightarrow \\ & (2+1)*(2+2) \end{aligned}$$

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 $w_1, w_2, \ldots$) 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 sind. 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 3.1.3. *Sei $G = (\{S\}, \{a\}, \{S \to aS\}, S)$. Dann gilt $L(G) = \emptyset$, denn es lassen sich keine Wörter aus $\{a\}^*$ ableiten: Die einzig mögliche Folge von Ableitungsschritten ist $S \Rightarrow aS \Rightarrow aaS \Rightarrow \ldots$, die jedoch nie endet. Im Vergleich dazu gilt für $G' = (\{S'\}, \{a, b\}, \{S' \to aS', S' \to b\}, S')$, dass $L(G') = \{a^n b \mid n \in \mathbb{N}\}$.*

Beispiel 3.1.4. *Als weiteres Beispiel betrachte die Grammatik $G = (\{S, B, C\}, \{a, b, c\}, P, S)$ mit*

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

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

$S \Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow aaaSBCBCBC \Rightarrow aaaaBCBCBCBC \Rightarrow aaaabCBCBCBC \Rightarrow aaaabBCCBCBC \Rightarrow aaaabbCCBCBC \Rightarrow aaaabbCBCCBC \Rightarrow aaaabbBCCCBC \Rightarrow aaaabbBCCBCC \Rightarrow aaaabbBCBCCC \Rightarrow aaaabbBBCCCC \Rightarrow aaaabbbBCCCC \Rightarrow aaaabbbbCCCC \Rightarrow aaaabbbbcCCC \Rightarrow aaaabbbbccCC \Rightarrow aaaabbbbcccC \Rightarrow aaaabbbbcccc.$

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 $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N}_{>0}\}$, was wir nun beweisen (⋆): Wir müssen zwei Richtungen zeigen

- *Teil 1: Zeige $a^n b^n c^n \in L(G)$ für alle $n \in \mathbb{N}_{>0}$: Wende $n-1$ mal die Regel $S \to aSBC$ und dann einmal die Regel $S \to aBC$ an. Das ergibt $S \Rightarrow^* a^n(BC)^n$. Nun wende die Regel $CB \to BC$ solange an, bis es kein Teilwort CB mehr gibt. Danach müssen alle Vorkommen von C rechts der Vorkommen von B stehen, daher gilt $a^n(BC)^n \Rightarrow^* a^nB^nC^n$. Wende Regel $aB \to ab$ und anschließend $n-1$ mal $bB \to bb$ an. Das ergibt $a^nB^nC^n \Rightarrow^* a^nb^nC^n$. Schließlich wende einmal $bC \to bc$ und anschließend $n-1$ mal $cC \to cc$ an, sodass $a^nb^nC^n \Rightarrow^* a^nb^nc^n$. Zusammensetzen aller Ableitungsschritte zeigt $S \Rightarrow^* a^nb^nc^n$.*
- *Teil 2: Zeige, dass alle von G erzeugten Wörter von der Form $a^nb^nc^n$ 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^nw' wobei in $\#_b(w') + \#_B(w') = \#_c(w') + \#_C(w') = n$.*
  - *Sei $S \Rightarrow^* a^nw$ mit $w \in \{b, c\}^*$. Dann muss das erste Zeichen von w ein b sein (denn ein auf a folgendes Zeichen kann nur durch $aB \to ab$ erzeugt werden, und die Regeln vertauschen keine Terminalsymbole). Ebenso kann das linkeste c nur durch die Regel $bC \to bc$ erzeugt sein. Alle andere Terminale b und c können nur durch $bB \to bb$ und $bC \to bc$*

und $cC \to cc$ 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 3.1.5. *Als weiteres Beispiel betrachte die Grammatik $G = (\{S, T, A, B, \$\}, \{a, b\}, P, S)$ mit*

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

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 $S \to \$T\$$ erzeugt zunächst eine Umrahmung mit \$\$. Anschließend erzeugen die drei Regeln $T \to aAT$, $T \to bBT$ und $T \to \varepsilon$ ein Wort aus 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 aus a und b platzieren. Dies geschieht dadurch das zunächst mit den Regeln $Aa \to aA, Ab \to bA, Ba \to aB, Bb \to bB$ die A's und B's bis vor das rechte \$ geschoben werden. Dann werden mit $A\$ \to \$a$ und $B\$ \to \$b$ die A's und B's in a's und b's verwandelt, wobei sie dabei über rechte \$ hüpfen, und dadurch quasi getrennt aufbewahrt werden. Mit den Regeln $\$a \to a\$$, $\$b \to b\$$ und $\$\$ \to \varepsilon$ wird das linke \$-Symbol zum rechten hin geschoben und schließlich werden beide \$ eliminiert. Bei allen Schritten wird die relative Lage aller a und b sowie aller A und B nicht geändert.

3.2. Die Chomsky-Hierarchie

Von Noam Chomsky wurde die folgende Einteilung der Grammatiken in Typen 0 bis 3 vorgenommen:

Definition 3.2.1 (Chomsky-Hierarchie)**.** *Es gilt die folgende Einteilung von Grammatiken (siehe Definition 3.1.1) 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 $(\ell \to r) \in P$ 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 $(\ell \to r) \in P$ 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 $(A \to r) \in P$ 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 3.2.2. *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 Namen kontextfrei und kontextsensitiv rühren daher, dass bei kontextfreien Grammatiken ein Vorkommen einer Variablen A immer (ohne Beschränkung) durch r ersetzt werden kann, wenn es eine Produktion $A \to r$ gibt. Bei kontextsensitiven Grammatiken können die Produktionen diese Ersetzung auf einen bestimmten Kontext einschränken: Z.B. kann durch die Produktion $uAv \to urv$ zugesichert werden, dass A nur dann durch r ersetzt wird, wenn es umrahmt von u und v vorkommt (diese Umrahmung ist der „Kontext“).

Beispiel 3.2.3. *Die Grammatik aus Beispiel 3.1.2 ist kontextfrei (vom Typ 2), während die Grammatik aus Beispiel 3.1.4 kontextsensitiv (vom Typ 1) ist. Die Grammatiken G und G' aus Beispiel 3.1.3 sind regulär (vom Typ 3). Die Grammatik aus Beispiel 3.1.5 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 Worts der Länge n daher je nach Typ der Grammatik wie in Abb. 3.1 veranschaulicht werden (vergl. (Sch08, S.10)).

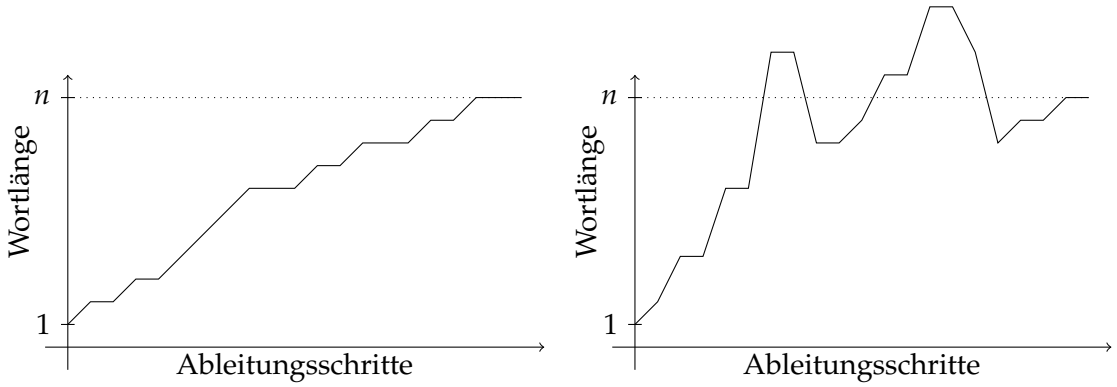


Abbildung 3.1.: Illustrationen zur Ableitung eines Wortes der Länge n mit einer Typ 1-Grammatik (links) und einer Typ 0-Grammatik (rechts)

3.2.1. Erzeugung des leeren Worts und ε-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 3.2.4 (1. Sonderregel: ε-Produktionen in Typ 1,2,3-Grammatiken)**.** *Eine Grammatik $G = (V, \Sigma, P, S)$ vom Typ 1, 2 oder 3 darf eine Produktion $(S \to \varepsilon) \in P$ enthalten, vorausgesetzt, dass keine rechte Seite einer Produktion in P, die Variable S enthält.*

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

Satz 3.2.5. *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 $S' \to \varepsilon$ hinzugefügt wird und alle Produktionen für S für das neue Startsymbol S' kopiert werden, d.h.*

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

Dann ist G' vom Typ i (mit der 1. Sonderregel gemäß Definition 3.2.4) und $L(G') = L(G) \cup \{\varepsilon\}$.

(⋆) Beweis. Da S' neu ist, kommt S' auf keiner rechten Seite vor. Alle Produktionen aus G' erfüllen die Anforderungen an eine Typ i Grammatik, da die Regeln aus G diese erfüllen. Da $S' \Rightarrow \varepsilon$, gilt $\varepsilon \in L(G')$. Für alle Ableitungen $S \Rightarrow_G^* w$ gibt es auch eine Ableitung $S' \Rightarrow_G^* w$ (verwende nur im ersten Ableitungsschritt die kopierte Produktion $S' \to r$ anstelle der Produktion $S \to r$). Ebenso gibt es für alle Ableitungen $S' \Rightarrow_{G'}^* w \neq \varepsilon$ eine Ableitung $S \Rightarrow_G w$ (verwende statt der Kopie $S' \to r$ stets die ursprüngliche Regel $S \to r$). □

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

Definition 3.2.6 (2. Sonderregel: ε-Produktionen in Typ 2- und Typ 3-Grammatiken)**.** *Eine Grammatik $G = (V, \Sigma, P, S)$ des Typs 2 oder 3 darf Produktionen von der Form $A \to \varepsilon$ enthalten, wo $A \in V \setminus \{S\}$.*

Satz 3.2.7 (Entfernen von ε-Produktionen in kontextfreien Grammatiken)**.** *Sei $G = (V, \Sigma, P, S)$ eine kontextfreie (bzw. reguläre) Grammatik. Dann gibt es eine kontextfreie (bzw. reguläre) Grammatik G' mit $L(G) = L(G')$ und G' enthält keine ε-Produktionen außer möglicherweise $S \to \varepsilon$.*

(⋆) Beweis. Die Konstruktion der Grammatik G' geschieht durch Algorithmus 1.
Die letzte Wiederhole-Schleife ist der interessante Teil des Algorithmus. Hier wird die Regelanwendung $A \to \varepsilon$ 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 $A' \to a$ sein müssen und daher dem Format der regulären Grammatiken entsprechen. D.h. G' ist in diesem Fall ebenfalls regulär. □

Beispiel 3.2.8 ((⋆) Entfernen von ε-Produktionen)**.** *Wir entfernen die ε-Produktion der Grammatik $G = (\{A, B, C, D, S\}, \{0, 1\}, P, S)$ mit*

$$P = \{S \to 1A,\ A \to AB,\ A \to DA,\ A \to \varepsilon,\ B \to 0,\ B \to 1,\ C \to AAA,\ D \to 1AC\}.$$

Algorithmus 1 : (⋆) Entfernen von ε-Produktionen

Eingabe : Typ i-Grammatik $G = (V, \Sigma, P, S)$ mit ε-Produktionen, $i \in \{2, 3\}$
Ausgabe : Typ i-Grammatik G' ohne ε-Produktionen (außer $S \to \varepsilon$, falls vorhanden), sodass $L(G) = L(G')$

Beginn
  finde die Menge $W \subseteq V$ aller Variablen A für die gilt $A \Rightarrow^* \varepsilon$:
  Beginn
    $W := \{A \mid (A \to \varepsilon) \in P \text{ und } A \neq S\}$;
    wiederhole
      füge alle solchen A zu W hinzu, für die es eine Produktion $A \to A_1 \dots A_n$ gibt, sodass für alle $i = 1, \dots, n$: $A_i \in W$;
    bis *sich W nicht mehr ändert*;
  Ende
  $P' := P \setminus \{A \to \varepsilon \mid (A \to \varepsilon) \in P \text{ und } A \neq S\}$; /* lösche Regeln $A \to \varepsilon$ */
  wiederhole
    für alle *Produktionen der Form $A' \to uAv$ in P' mit $|uv| > 0$ und $A \in W$* **tue**
      füge die Produktion $A' \to uv$ zu P' hinzu;
      /* für eine Produktion $A' \to u'Av'Aw'$ 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*;
  Gib $G' = (V, \Sigma, P', S)$ als Ergebnisgrammatik aus;
Ende

Die Menge der Variablen, die ε herleiten, ist $W = \{A, C\}$ (zunächst wird A eingefügt, da es die Produktion $A \to \varepsilon$ gibt, danach wird C eingefügt, da $C \to AAA$). Löschen der Produktion $A \to \varepsilon$ ergibt $P' = \{S \to 1A, A \to AB, A \to DA, B \to 0, B \to 1, C \to AAA, D \to 1AC\}$. Hinzufügen der Regeln (Löschen von Vorkommen von A, C) ergibt

$$P' = \{S \to 1A,\ S \to 1,\ A \to AB,\ A \to B,\ A \to DA,\ A \to D,\ B \to 0,\ B \to 1,\\ C \to AAA,\ C \to AA,\ C \to A,\ D \to 1AC,\ D \to 1A,\ D \to 1C,\ D \to 1\}.$$

Daher ist $G' = (\{A, B, C, D, S\}, \{0, 1\}, P', S)$.

3.2.2. 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:

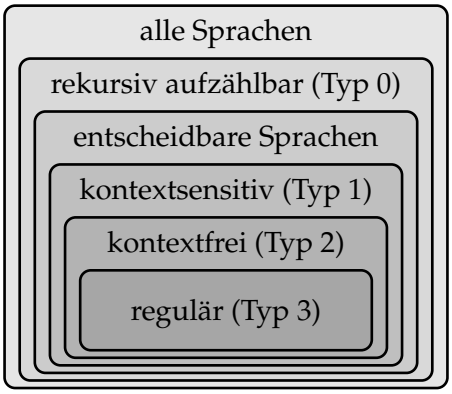
$$\text{Typ 3-Sprachen} \subset \text{Typ 2-Sprachen} \subset \text{Typ 1-Sprachen} \subset \text{Typ 0-Sprachen}.$$

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

Eine Sprache L heißt *entscheidbar*, wenn es einen Algorithmus gibt, der bei Eingabe eines Wortes w in endlicher Zeit feststellen kann, ob $w \in L$ gilt oder nicht. Die Typ 1-, 2- und 3-Sprachen sind entscheidbar, während es Typ 0-Sprachen gibt, die nicht entscheidbar sind (für obige Sprache H trifft dies zu). Typ 0-Sprachen sind jedoch rekursiv aufzählbar (oder semi-entscheidbar), d.h. für jede Typ 0-Sprache L gibt es einen Algorithmus, der bei Eingabe eines Wortes $w \in L$ in endlicher Zeit feststellt, dass $w \in L$ gilt, und bei einem Wort $w \notin L$ entweder feststellt, dass $w \notin L$ gilt, oder nicht terminiert.

Die Menge der Typ 0-Grammatiken ist abzählbar, da jede Grammatik eine endliche Beschreibung hat und die Grammatiken daher der Größe nach aufgezählt werden können. Hingegen ist die Menge aller Sprachen überabzählbar und hat dieselbe Kardinalität wie die reellen Zahlen.

Daher ergibt sich folgendes Bild:



Für 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 zuzüglich 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 Ausdrücken, u.s.w. sind nicht mit einer kontextfreien Sprache darstellbar). Dennoch wird die Syntax von Programmiersprachen oft durch kontextfreie Grammatiken beschrieben und zusätzliche Nebenbedingungen (wie korrekte Typisierung) darüber hinaus festgelegt.

3.3. Das Wortproblem

Sei $S \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_m$ eine Ableitung des Wortes w_m der Länge n in einer kontextsensitiven Grammatik. Da $|\ell| \leq |r|$ für alle Produktionen $\ell \to r$ 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 3.3.1 (Wortproblem für Typ i-Grammatiken)**.** *Das Wortproblem für Typ i-Grammatiken 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 3.3.2. *Das Wortproblem für Typ 1-Grammatiken 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 $m \in \mathbb{N}, n \in \mathbb{N}_{>0}$ 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 lassen sich rekursiv wie folgt berechnen:

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

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 $(|\Sigma \cup V| + 1)^n$ 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 $k \in \mathbb{N}$. 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 $k \in \mathbb{N}$ 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 2 das Wortproblem für Typ 1-Grammatiken. □

Algorithmus 2 : (⋆) Entscheiden des Wortproblems für Typ 1-Grammatiken

Eingabe : Typ 1-Grammatik $G = (V, \Sigma, P, S)$ (mit 1. Sonderregel) 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
    $L_{\text{old}} := L$;
    $L := \mathit{next}(L_{\text{old}}, n)$;
  bis *($w \in L$) oder ($L_{old} = L$)*;
  wenn $w \in L$ **dann**
    return *Ja*;
  sonst
    return *Nein*;
  Ende
Ende

Korollar 3.3.3. *Das Wortproblem für Typ 2-Grammatiken und das Wortproblem für Typ 3-Grammatiken sind jeweils entscheidbar.*

Beispiel 3.3.4 (⋆). *Als Beispiel betrachte die Grammatik* $G = (\{S, B\}, \{a, b, c\}, P, S)$ *mit* $P = \{S \to aSBc, S \to abc, cB \to Bc, bB \to bb\}$. *Wir berechnen* L_m^6 *für alle m:*

$$
\begin{aligned}
L_0^6 &= \{S\} \\
L_1^6 &= next(L_0^6) = L_0^6 \cup \{aSBc, abc\} = \{S, aSBc, abc\} \\
L_2^6 &= next(L_1^6) = L_1^6 \cup \{aSBc, abc, aabcBc\} = \{S, aSBc, abc, aabcBc\} \\
L_3^6 &= next(L_2^6) = L_2^6 \cup \{aSBC, abc, aabcBc, aabBcc\} = \{S, aSBc, abc, aabcBc, aabBcc\} \\
L_4^6 &= next(L_3^6) = L_3^6 \cup \{aSBc, abc, aabcBc, aabBcc, aabbcc\} \\
&\quad = \{S, aSBc, abc, aabcBc, aabBcc, aabbcc\} \\
L_5^6 &= next(L_4^6) = L_4^6 \cup \{aSBc, abc, aabcBc, aabBcc, aabbcc\} \\
&\quad = \{S, aSBc, abc, aabcBc, aabBcc, aabbcc\}
\end{aligned}
$$

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.*

3.4. Weitere Probleme für formale Sprachen

Neben dem Wortproblem existieren andere Fragestellungen für formale Sprachen, die wir in den folgenden Definitionen spezifizieren.

Definition 3.4.1. *Das Leerheitsproblem für Sprachen vom Typ i ist die Frage, ob für eine Typ i-Grammatik G, die Gleichheit* $L(G) = \emptyset$ *gilt.*

Definition 3.4.2. *Das Endlichkeitsproblem für Sprachen vom Typ i ist die Frage, ob für eine Typ i-Grammatik G die Ungleichheit* $|L| < \infty$ *gilt.*

Definition 3.4.3. *Das Schnittproblem für Sprachen vom Typ i ist die Frage, ob für Typ i-Grammatiken* G_1, G_2 *gilt:* $L(G_1) \cap L(G_2) = \emptyset$.

Definition 3.4.4. *Das Äquivalenzproblem für Sprachen vom Typ i ist, die Frage, ob Typ i-Grammatiken* G_1, G_2 *gilt:* $L(G_1) = L(G_2)$.

Die Entscheidbarkeit dieser Probleme werden wir in späteren Kapitel für die unterschiedlichen Typen von Sprachen betrachten.

3.5. Syntaxbäume

Wir haben Syntaxbäume bereits verwendet und definieren diese nun formal:

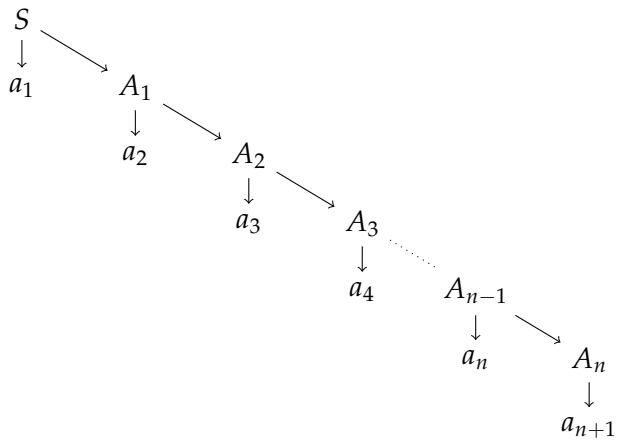
Definition 3.5.1. *Sei* $G = (V, \Sigma, P, S)$ *eine Typ 2-Grammatik und* $S \Rightarrow w_0 \Rightarrow \ldots \Rightarrow w_n$ *eine Ableitung von* $w_n \in \Sigma^*$. *Der* Syntaxbaum *zur Ableitung wird wie folgt erstellt:*

- *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 angewandte Produktion ist $A \to r$), 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.

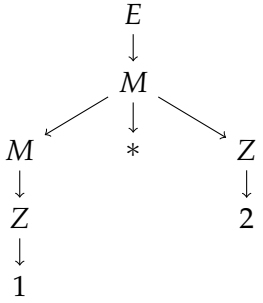
Beachte, dass für Typ-3-Grammatiken, Syntaxbäume immer eine listenartige Struktur folgender Form haben



Beispiel 3.5.2. *Wir betrachten erneut die Grammatik $G = (\{E, M, Z\}, \{+, *, 1, 2, (,)\}, P, E)$ mit*

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

(siehe Beispiel 3.1.2). 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:

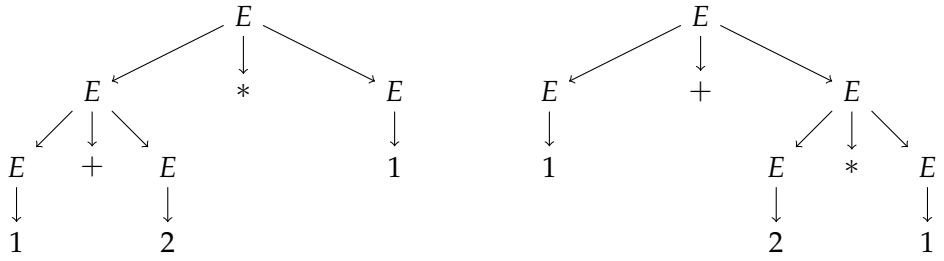
Satz 3.5.3. *Sei G eine Typ 2-Grammatik und $w \in L(G)$. Dann gibt es eine Linksableitung (und eine Rechtsableitung) von w.*

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

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

Eine Ableitung des Worts $1 + 2 * 1$ ist $\underline{E} \Rightarrow \underline{E} * E \Rightarrow \underline{E} + E * E \Rightarrow 1 + \underline{E} * E \Rightarrow 1 + 2 * \underline{E} \Rightarrow 1 + 2 * 1$ eine andere Ableitung ist $\underline{E} \Rightarrow E + \underline{E} \Rightarrow \underline{E} + E * E \Rightarrow 1 + \underline{E} * E \Rightarrow 1 + 2 * \underline{E} \Rightarrow 1 + 2 * 1$. 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 $\{a^m b^m c^n d^n \mid m, n \in \mathbb{N}_{>0}\} \cup \{a^m b^n c^n d^m \mid m, n \in \mathbb{N}_{>0}\}$ (siehe (HMU06, Abschnitt 5.4.4)).

3.6. 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 3.6.1 (Erweiterte Backus-Naur-Form (EBNF))**.** *Für Typ 2-Grammatiken erlauben wir die folgenden abkürzenden Schreibweisen für die Menge der Produktionen P:*

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

4. 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 nicht-deterministische 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 und den Satz von Myhill und Nerode 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.

4.1. Deterministische endliche Automaten

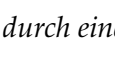
Endliche Automaten lesen als Eingabe ein Wort (schrittweise) und *akzeptieren* oder *verwerfen* das Wort. Die akzeptierte Sprache eines solchen Automaten besteht aus den Wörtern, die von ihm akzeptiert werden.

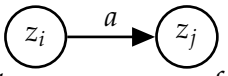
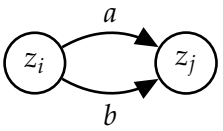
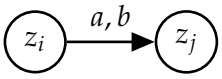
Definition 4.1.1 (Deterministischer Endlicher Automat, DFA)**.** *Ein* deterministischer endlicher Automat *(deterministic finite automaton, DFA) ist ein 5-Tupel* $M = (Z, \Sigma, \delta, z_0, E)$*, wobei*

- Z *ist eine endliche Menge von* Zuständen*,*
- Σ *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*
- $\delta : Z \times \Sigma \to Z$ *ist die* Zustandsüberführungsfunktion *(oder nur* Überführungsfunktion*).*

Deterministische endliche Automaten können durch Zustandsgraphen dargestellt werden:

Definition 4.1.2. *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*

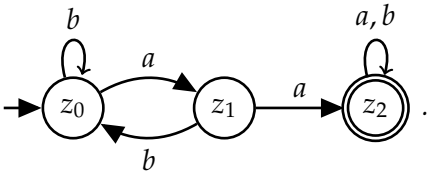
- *Zustände* $z \in Z$ *als* *gezeichnet werden,*
- *der Startzustand* $z_0 \in Z$ *durch einen auf ihn zeigenden Pfeil markiert wird, d.h.*  *,*
- *Endzustände* $z \in E$ *mit doppelten Kreisen markiert werden, d.h. als* *,*

- *für $\delta(z_i, a) = z_j$, je eine mit a beschriftete Kante von z_i nach z_j*  *gezeichnet wird. Dabei werden gleiche Kanten mit unterschiedlicher Beschriftung zusammengefasst, indem nur eine Kante gezeichnet wird, und die Beschriftungen, mit Kommas getrennt an die Kante geschrieben werden: Anstelle von*  *zeichnen wir*  .

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

Beispiel 4.1.3. *Der DFA $M = (\{z_0, z_1, z_2\}, \{a, b\}, \delta, z_0, \{z_2\})$ mit*

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

hat den Zustandsgraph  .

Die Abarbeitung eines Wortes mit einem Automaten kann man sich 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 Überführungsfunktion δ) gewechselt. Ist das Wort komplett eingelesen und der aktuelle Zustand ist ein Endzustand, dann wird das Wort vom DFA *erkannt*.

Beispiel 4.1.4. *Betrachte den DFA M aus Beispiel 4.1.3. 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 4.1.5 (Akzeptierte Sprache eines DFAs)**.** *Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA. Wir definieren die Funktion $\widetilde{\delta} : Z \times \Sigma^* \to Z$ durch*

$$\widetilde{\delta}(z, \varepsilon) := z \;\; \textit{und} \;\; \widetilde{\delta}(z, aw) := \widetilde{\delta}(\delta(z, a), w)$$

Die von M akzeptierte Sprache *ist*

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

Beachte, dass $\widetilde{\delta}$ die Überführungsfunktion δ solange anwendet, bis das Wort abgearbeitet wurde, d.h. $\widetilde{\delta}(z, a_1 \cdots a_n) = \delta(\ldots\delta(\delta(z, a_1), a_2), \ldots, a_n)$.
Lässt man den Automaten für ein Wort ablaufen, so spricht auch von einem *Lauf* des DFAs:

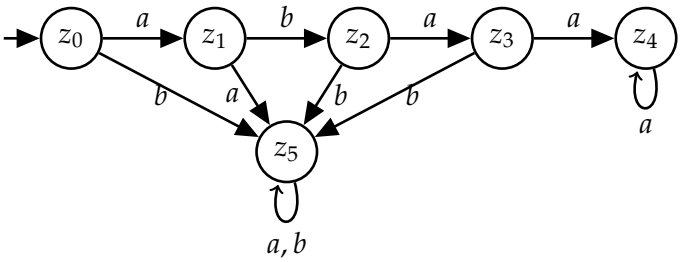
Definition 4.1.6. *Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA und $w \in \Sigma^*$ mit $|w| = n$. Die Folge von Zuständen $q_0, \ldots, q_n$ mit $q_0 = z_0$ und $q_i = \delta(q_{i-1}, w[i])$ für $1 \leq i \leq n$ bezeichnet man als* Lauf von M für Wort w. *Für einen solchen Lauf schreiben wir auch:*

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

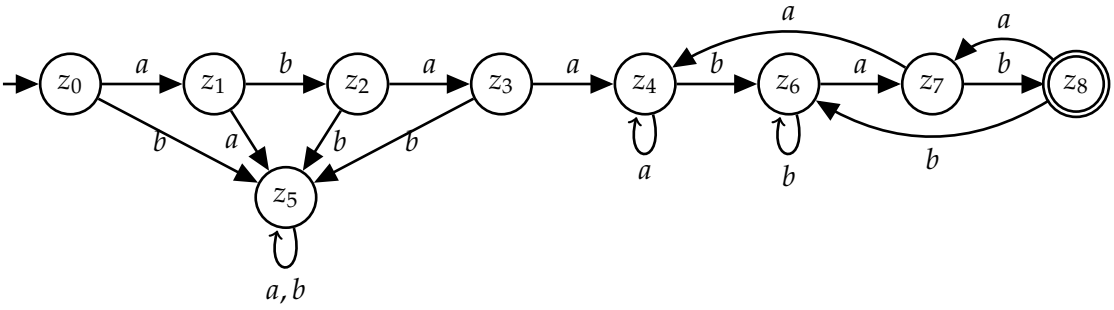
Ein Lauf der mit einem Endzustand endet, nennen wir auch akzeptierender Lauf.

Beispiel 4.1.7. *Für den DFA M aus Beispiel 4.1.3 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, indem man beobachtet, dass z_2 von z_0 aus nur über aa erreicht werden kann, dass nach Erreichen von z_2 in z_2 verblieben wird, und dass man nach Lesen von b in z_1 wieder erneut in z_0 starten muss.*

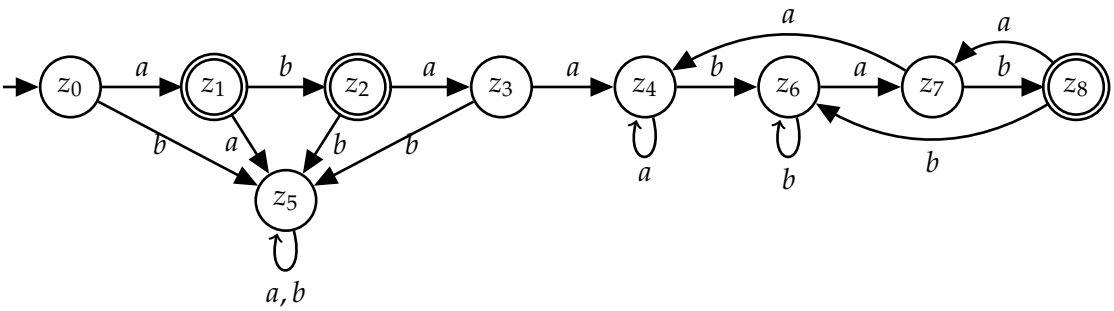
Beispiel 4.1.8. *Wir konstruieren einen DFA über $\Sigma = \{a, b\}$, der die Sprache aller Wörter akzeptiert, die mit abaa beginnen und mit bab enden: Für den Präfix abaa können wir vier Zustände z_1, z_2, z_3, z_4. (zusätzlich zum Startzustand z_0) erzeugen, die vom Startzustand durchlaufen werden müssen, und einen weiteren Zustand z_5, der zum Fehler führt (falls das Präfix nicht stimmt). Das ergibt bereits:*



Für das 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 das Suffix bab noch kommt bzw. komplettiert wird: Wenn wir in z_4 ein a lesen, verbleiben wir in z_4, da das 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 das gesamte 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 das geforderte 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 das geforderte Suffix noch ein b lesen, daher wechseln wir in z_7:



Beispiel 4.1.9. *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 4.1.8 konstruiert werden, wobei z_1 und z_2 zusätzliche Endzustände sind, d.h. der folgende Automat akzeptiert die genannte Sprache:*



Übungsaufgabe 4.1.10. *Ein Kaugummi-Automat erhält als Eingabe 10- und 20-Cent Münzen und akzeptiert g.d.w. 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.*

4.2. 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 4.2.1. *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 $z_i \to az_j \in P$ und falls $z_j \in E$ zusätzlich die Produktion $z_j \to \varepsilon$ (mit Hilfe der ϵ-Regeln).
Wie zeigen, dass für jedes Wort $w \in \Sigma^*$ gilt: $w \in L(M) \iff w \in L(G)$. Falls $w = a_1 \cdots a_m$, dann gilt $w \in L(M)$ g.d.w. es Zustände $z_1, \ldots, z_m \in Z$ gibt mit $\delta(z_{i-1}, a_i) = z_i$ und $z_m \in E$. Die letzte Aussage ist äquivalent dazu, dass es Ableitungsschritte $a_1 \cdots a_{i-1} z_{i-1} \Rightarrow_G a_1 \cdots a_i z_i$ für $i = 1, \ldots, m$ und $a_1 \cdots a_m z_m \Rightarrow_G a_1 \cdots a_m$ 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. □

Beispiel 4.2.2. *Die reguläre Grammatik zum DFA aus Beispiel 4.1.3 entsprechend zum Beweis von Theorem 4.2.1 ist $G = (V, \Sigma, P, S)$ mit $V = \{z_0, z_1, z_2\}$, $\Sigma = \{a, b\}$, $S = z_0$ und*

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

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

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

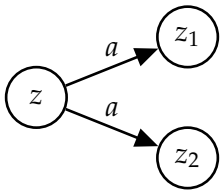
Die Umkehrung der letzten Aussage (für jede reguläre Sprache L gibt es einen DFA M, der L akzeptiert) gilt auch, aber wir brauchen weitere Mittel, um sie zu beweisen. Die Kodierung im Beweis der letzten Aussage ist nicht immer rückwärts anwendbar, sobald es z.B. Produktionen $A \to aA_1$ und $A \to aA_2$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

Ein Hilfsmittel sind daher die nichtdeterministischen endlichen Automaten, die wir im nächsten Abschnitt einführen werden.

4.3. Nichtdeterministische endliche Automaten

Nichtdeterministische Automaten ermöglichen es dem Automaten, nicht eindeutig, sondern durch „Raten“ in einen Zustand zu wechseln. Während die Überführungsfunktion δ 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 Überführungsfunktion δ 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 4.3.1. *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*,*
- *Σ 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*
- *$\delta : Z \times \Sigma \to \mathcal{P}(Z)$ ist die* Zustandsüberführungsfunktion *(oder nur* Überführungsfunktion*).*

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 4.3.2 (Akzeptierte Sprache eines NFA). *Sei $M = (Z, \Sigma, \delta, S, E)$ ein NFA. Wir definieren $\widetilde{\delta} : (\mathcal{P}(Z) \times \Sigma^*) \to \mathcal{P}(Z)$ rekursiv durch*

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

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

Beachte, dass die Bedingung $\widetilde{\delta}(S, w) \cap E \neq \emptyset$ äquivalent dazu ist, dass ein Endzustand $z \in E$ existiert, sodass $z \in \widetilde{\delta}(S, w)$ (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:

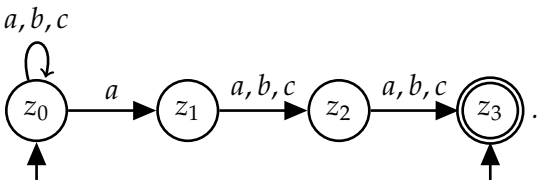
Beispiel 4.3.3. *Sei $M = (Z, \Sigma, \delta, \emptyset, E)$ ein NFA. Dann ist $L(M) = \emptyset$.*

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

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

Der Zustandsgraph zu M ist



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. $L(M) = \{\varepsilon\} \cup (\{a, b, c\}^* \cdot \{a\} \cdot \{a, b, c\} \cdot \{a, b, c\})$ 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.

4.4. Reguläre Sprachen können durch NFAs erkannt werden

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

Theorem 4.4.1. *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 $\delta : Z \times \Sigma \to \mathcal{P}(Z)$ definiert durch $\delta(A, a) := \{B \mid A \to aB \in P\} \cup \{z_E \mid \text{falls } A \to a \in P\}$ für alle $A \in V$ und $a \in \Sigma$ und $\delta(z_E, a) = \emptyset$ für alle $a \in \Sigma$. Falls $S \to \varepsilon \in P$ sei $E = \{S, z_E\}$, und ansonsten sei $E = \{z_E\}$.

Wir zeigen, dass $L(M) = L(G)$ gilt. Offensichtlich gilt $\varepsilon \in L(M)$ g.d.w. $\varepsilon \in L(G)$. Für $w = a_1 \cdots a_n$ gilt: $w \in L(G)$ g.d.w. es 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 $A_1, \ldots, A_{n-1}$ gibt, sodass $A_1 \in \delta(S, a_1)$, $A_{i+1} \in \delta(A_i, a_{i+1})$ für $i = 1, \ldots, n-2$ und $z_E \in \delta(A_{n-1}, a_n)$, was wiederum äquivalent dazu ist, dass $w \in L(M)$ gilt. □

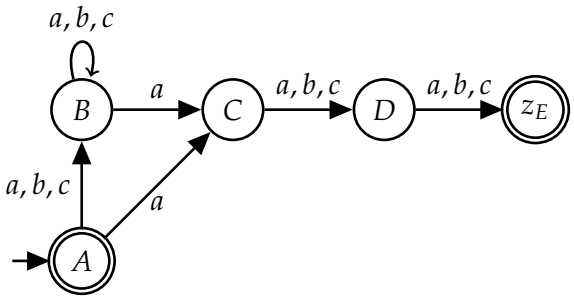
Beispiel 4.4.2. *Betrachte die reguläre Grammatik $G = (V, \Sigma, P, A)$ mit $V = \{A, B, C, D\}$, $\Sigma = \{a, b, c\}$ und*

$$\begin{aligned} P = \{&A \to \varepsilon \mid aB \mid bB \mid cB \mid aC, \\ &B \to aB \mid bB \mid cB \mid aC, \\ &C \to aD \mid bD \mid cD, \\ &D \to a \mid b \mid c\} \end{aligned}$$

Der zu G passende NFA gemäß der Konstruktion im Beweis von Theorem 4.4.1 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

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

Der Zustandsgraph zu M ist



Z.B. wird bacabc von G erzeugt (denn $A \Rightarrow bB \Rightarrow baB \Rightarrow bacB \Rightarrow bacaC \Rightarrow bacabD \Rightarrow bacabc$) 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 4.3.4 vom NFA M akzeptiert bzw. von der Grammatik G erzeugt.

4.5. 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 4.5.1. *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
- $\delta'(X, a) = \bigcup_{z \in X} \delta(z, a) = \widetilde{\delta}(X, a)$ (d.h. $\delta'(X, a)$ berechnet die Menge aller von X aus erreichbaren Zustände in M, was dasselbe ist, was $\widetilde{\delta}$ berechnet).

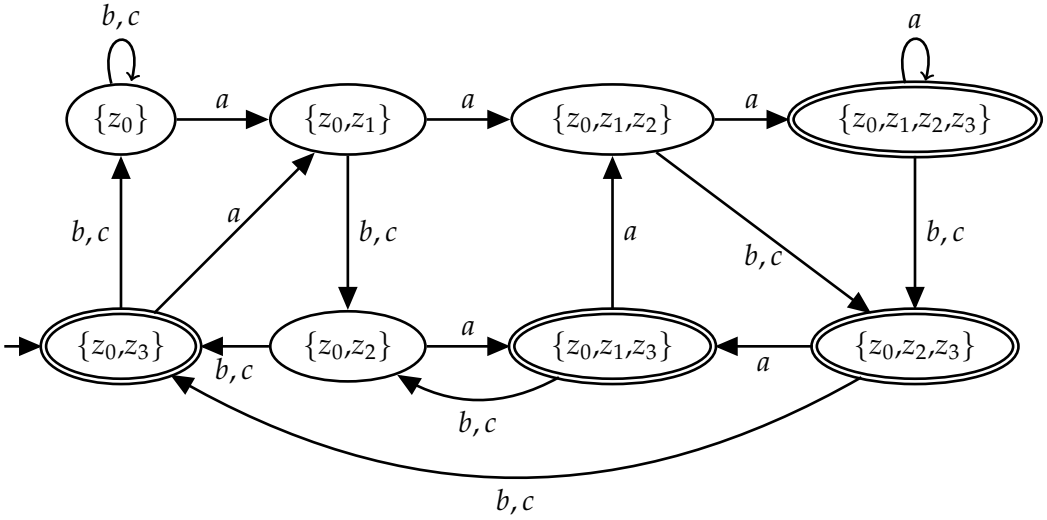
Wir beweisen, dass $L(M') = L(M)$ gilt. Sei $w = a_1 \cdots a_n \in \Sigma^*$.

Es gilt $w \in L(M)$ g.d.w. $\widetilde{\delta}(S, w) \cap E \neq \emptyset$. Gemäß Definition 4.3.2 ist äquivalent dazu, dass es eine Folge $Z_1, \ldots, Z_n$ von Teilmengen von Z gibt, mit $\widetilde{\delta}(S, a_1) = Z_1$, $\widetilde{\delta}(Z_i, a_{i+1}) = Z_{i+1}$ für $i = 1, \ldots, n-1$ und $Z_n \cap E \neq \emptyset$. Gemäß obiger Konstruktion folgt, dass dies äquivalent dazu ist, dass es eine Folge $Z_1, \ldots, Z_n$ von Teilmengen von Z gibt, mit $\delta'(S, a_1) = Z_1$, $\delta'(Z_i, a_{i+1}) = Z_{i+1}$ für $i = 1, \ldots, n-1$ und $Z_n \cap E \neq \emptyset$. Die letzte Aussage ist äquivalent zu $\widetilde{\delta'}(S', w) \in E'$, was wiederum äquivalent zu $w \in L(M')$ ist. □

Beispiel 4.5.2. *Betrachte den NFA M aus Beispiel 4.3.4. Entsprechend der Konstruktion im Beweis zu Theorem 4.5.1 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* $d \in \{a, b, c\}$
  $\delta'(\{z_0\}, a) = \{z_0, z_1\}$
  $\delta'(\{z_0\}, d) = \{z_0\}$ *für* $d \in \{b, c\}$
  $\delta'(\{z_1\}, d) = \{z_2\}$ *für* $d \in \{a, b, c\}$
  $\delta'(\{z_2\}, d) = \{z_3\}$ *für* $d \in \{a, b, c\}$
  $\delta'(\{z_3\}, d) = \emptyset$ *für* $d \in \{a, b, c\}$
  $\delta'(\{z_0, z_1\}, a) = \{z_0, z_1, z_2\}$
  $\delta'(\{z_0, z_1\}, d) = \{z_0, z_2\}$ *für* $d \in \{b, c\}$
  $\delta'(\{z_0, z_2\}, a) = \{z_0, z_1, z_3\}$
  $\delta'(\{z_0, z_2\}, d) = \{z_0, z_3\}$ *für* $d \in \{b, c\}$
  $\delta'(\{z_0, z_3\}, a) = \{z_0, z_1\}$
  $\delta'(\{z_0, z_3\}, d) = \{z_0\}$ *für* $d \in \{b, c\}$
  $\delta'(\{z_1, z_2\}, d) = \{z_2, z_3\}$ *für* $d \in \{a, b, c\}$
  $\delta'(\{z_1, z_3\}, d) = \{z_2\}$ *für* $d \in \{a, b, c\}$
  $\delta'(\{z_2, z_3\}, d) = \{z_3\}$ *für* $d \in \{a, b, c\}$
  $\delta'(\{z_0, z_1, z_2\}, a) = \{z_0, z_1, z_2, z_3\}$
  $\delta'(\{z_0, z_1, z_2\}, d) = \{z_0, z_2, z_3\}$ *für* $d \in \{b, c\}$
  $\delta'(\{z_0, z_1, z_3\}, a) = \{z_0, z_1, z_2\}$
  $\delta'(\{z_0, z_1, z_3\}, d) = \{z_0, z_2\}$ *für* $d \in \{b, c\}$
  $\delta'(\{z_0, z_2, z_3\}, a) = \{z_0, z_1, z_3\}$
  $\delta'(\{z_0, z_2, z_3\}, d) = \{z_0, z_3\}$ *für* $d \in \{b, c\}$
  $\delta'(\{z_1, z_2, z_3\}, d) = \{z_2, z_3\}$ *für* $d \in \{a, b, c\}$
  $\delta'(\{z_0, z_1, z_2, z_3\}, a) = \{z_0, z_1, z_2, z_3\}$
  $\delta'(\{z_0, z_1, z_2, z_3\}, d) = \{z_0, z_2, z_3\}$ *für* $d \in \{b, c\}$

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



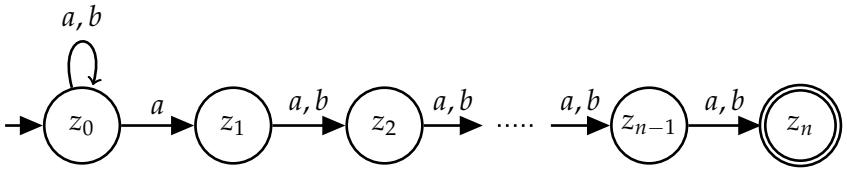
Die im Beweis zu Theorem 4.5.1 vorgestellte Potenzmengen-Konstruktion eines DFA aus einem NFA führt zu einer exponentiellen Anzahl an Zuständen im DFA bezüglich der Anzahl an Zuständen im NFA. Das Beispiel 4.5.2 zeigt, dass nicht alle Zustände tatsächlich benötigt werden (da unerreichbare entfernt werden können), aber es klärt auch nicht die Frage, ob man einen besseren Algorithmus angegeben kann, der einen DFA generiert, der mit weniger Zuständen auskommt.

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

Lemma 4.5.3 (⋆). *Sei $L_n = \{uav \mid u \in \{a,b\}^*, v \in \{a,b\}^{n-1}\}$ für $n \in \mathbb{N}_{>0}$, die Sprache aller Wörter aus $\{a,b\}^*$, die an n-letzter Stelle ein a haben. Dann gilt für alle $n \in \mathbb{N}_{>0}$:*

- *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 $z_0, z_1, \ldots z_n$ 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 $n \in \mathbb{N}_{>0}$ 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, sodass $\widetilde{\delta}(z_0, w) = \widetilde{\delta}(z_0, w') = z$ gilt

(nach Einlesen beider Wörter befindet sich M' in ein und demselben Zustand z). Sei j die erste Position an der sich w und w' unterscheiden. Wir erhalten o.B.d.A. $w = uav$ und $w' = ubv'$ mit $|u| = j-1$ und $|v| = |v'| = n-j$. Betrachte die Wörter $w_2 = uavb^{j-1}$ und $w_2' = ubv'b^{j-1}$ (wir verlängern w und w' um $j-1$ b's). Dann muss gelten $\widetilde{\delta}(w_2) = \widetilde{\delta}(w_2')$, da $\widetilde{\delta}(uav) = z = \widetilde{\delta}(ubv')$. Offensichtlich gilt aber $w_2 \in L_n$ und $w_2' \notin L_n$, was fordert, dass $\widetilde{\delta}(w_2) \in E$ und $\widetilde{\delta}(w_2') \notin E$, was ein Widerspruch ist (da $\widetilde{\delta}(w_2) = \widetilde{\delta}(w_2')$). Daher war unsere Annahme falsch und es gibt keinen DFA mit weniger als 2^n Zuständen, der die Sprache L_n akzeptiert. □

Theorem 4.5.4. *DFAs bzw. 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 4.2.1, (2) in Theorem 4.4.1 und (3) in Theorem 4.5.1 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. □

4.6. NFAs mit ε-Übergängen

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

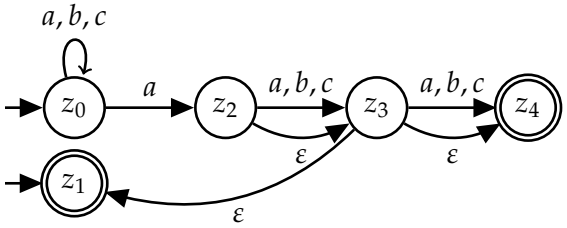
Ein NFA mit ε-Übergängen, erlaubt es von einem Zustand in einen anderen Zustand zu wechseln, *ohne* ein Zeichen zu lesen (sondern durch Lesen des leeren Worts ε). Diese Erweiterung führt nicht zu neuer Ausdruckskraft.

Definition 4.6.1 (NFA mit ε-Übergängen)**.** *Ein* nichtdeterministischer endlicher Automat mit ε-Übergängen *(NFA mit ε-Übergängen) ist ein Tupel $M = (Z, \Sigma, \delta, S, E)$, wobei*

- *Z ist eine endliche Menge von* Zuständen,
- *Σ 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*
- *$\delta : Z \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Z)$ ist die* Zustandsüberführungsfunktion *(oder nur* Überführungsfunktion*).*

Beachte, dass die Definition sich von der Definition der NFAs nur dadurch unterscheidet, dass der Definitionsbereich von δ 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 4.6.2. *Ein Beispiel für einen NFA mit ε-Übergängen ist:*



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

Definition 4.6.3 (ε-Hülle)**.** *Sei $M = (Z, \Sigma, \delta, S, E)$ ein NFA mit ε-Übergängen. Die ε-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 ε-Hülle für eine Zustandsmenge $X \subseteq Z$ kann auch berechnet werden durch:

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

Beispiel 4.6.4. *Betrachte den NFA mit ε-Übergängen aus Beispiel 4.6.2. Dann gilt:*

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

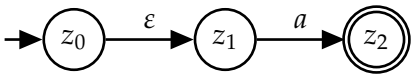
Für die Definition der akzeptierten Sprache eines NFA mit ε-Übergängen wird nun stets nach dem Anwenden der Überführungsfunktion δ die erhaltene Menge noch abgeschlossen bezüglich der ε-Übergänge (durch Anwenden des $clos_\varepsilon$-Operators).

Definition 4.6.5 (Akzeptierte Sprache eines NFA mit ε-Übergängen)**.** *Sei $M = (Z, \Sigma, \delta, S, E)$ ein NFA mit ε-Übergängen. Wir definieren $\widetilde{\delta} : (\mathcal{P}(Z) \times \Sigma^*) \to \mathcal{P}(Z)$ rekursiv durch*

$$\widetilde{\delta}(X, \varepsilon) := X \quad \textit{und} \quad \widetilde{\delta}(X, aw) := \widetilde{\delta}(\bigcup_{z \in X} clos_\varepsilon(\delta(z, a)), w) \textit{ für alle } X \subseteq Z$$

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

Bemerkung 4.6.6. *Die Anwendung des $clos_\varepsilon$-Operators auf die Menge S ist notwendig. Betrachte den NFA mit ε-Übergängen:*



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

Satz 4.6.7. *NFAs mit ε-Übergängen akzeptieren genau die regulären Sprachen.*

Beweis. Wir zeigen zuerst, dass jede reguläre Sprache von einem NFA mit ε-Übergängen akzeptiert wird: Sei L regulär. Dann gibt es einen NFA $M = (Z, \Sigma, \delta, S, E)$, der L akzeptiert. Der NFA mit ε-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 ε-Übergängen sind regulär) sei ein NFA mit ε-Übergängen $M = (Z, \Sigma, \delta, S, E)$ gegeben. Wir konstruieren einen NFA M' mit $L(M') = L(M)$, woraus dann mit Theorem 4.5.4 direkt folgt, dass $L(M)$ regulär ist.

Sei $M' = (Z, \Sigma, \delta', S', E)$ mit $S' = clos_\varepsilon(S)$ und $\delta'(z,a) = clos_\varepsilon(\delta(z,a))$.

Wir zeigen, dass für alle $X \subseteq Z$ und alle $w \in \Sigma^*$ gilt: $\widetilde{\delta}(clos_\varepsilon(X), w) = \widetilde{\delta'}(clos_\varepsilon(X), w)$. 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 $\widetilde{\delta}(clos_\varepsilon(X), \varepsilon) = clos_\varepsilon(X) = \widetilde{\delta'}(clos_\varepsilon(X), \varepsilon)$.

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

Wir formen um:

$$\begin{aligned}\widetilde{\delta}(clos_\varepsilon(X), au) &= \widetilde{\delta}(\bigcup_{z \in clos_\varepsilon(X)} clos_\varepsilon(\delta(z,a)), u) =_{I.H.} \widetilde{\delta'}(\bigcup_{z \in clos_\varepsilon(X)} clos_\varepsilon(\delta(z,a)), u) \\ &= \widetilde{\delta'}(\bigcup_{z \in clos_\varepsilon(X)} \delta'(z,a), u) = \widetilde{\delta'}(clos_\varepsilon(X), au) = \widetilde{\delta'}(clos_\varepsilon(X), w)\end{aligned}$$

□

Für NFAs mit ε-Übergängen können wir leicht fordern, dass sie genau einen Startzustand und genauen einen Endzustand besitzen:

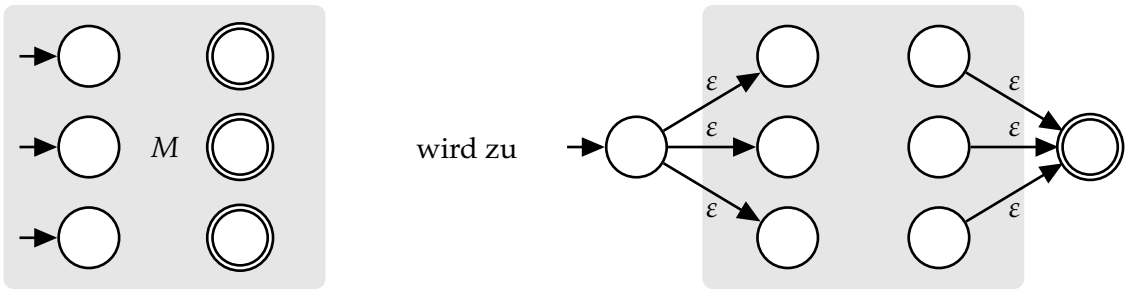
Satz 4.6.8 (NFA mit ε-Übergängen und eindeutigen Start- und Endzuständen)**.** *Für jeden NFA M mit ε-Übergängen gibt es einen NFA M' mit ε-Übergängen, sodass $L(M') = L(M)$ und M' genau einen Startzustand und genau einen Endzustand hat, wobei diese beiden Zustände verschieden sind.*

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

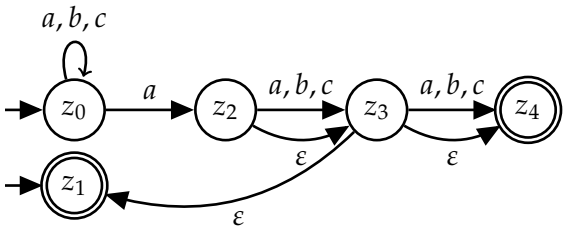
$$\begin{aligned}
\delta'(z_0, \varepsilon) &= S \\
\delta'(z_0, a) &= \emptyset && \text{für alle } a \in \Sigma \\
\delta'(z, w) &= \delta(z, w) && \text{für alle } z \in Z \text{ und } w \in (\{\varepsilon\} \cup \Sigma) \\
\delta'(z, \varepsilon) &= \{z_E\} && \text{für alle } z \in E \\
\delta'(z_E,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

Bildlich dargestellt:

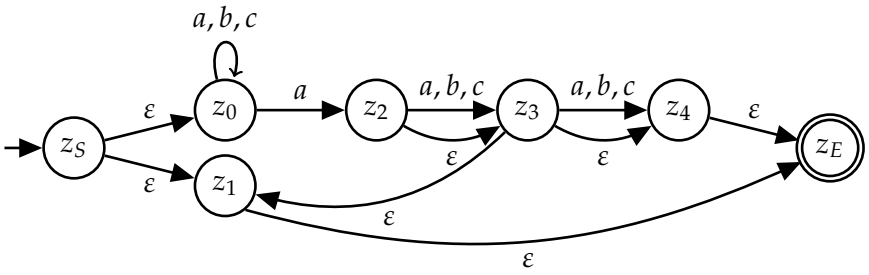


Offensichtlich gilt $L(M') = L(M)$. □

Beispiel 4.6.9. *Betrachte erneut den NFA mit ε-Übergängen aus Beispiel 4.6.2:*



Der dazu entsprechend Satz 4.6.8 konstruierte NFA mit ε-Übergängen und eindeutigen Start- und Endzuständen ist:



4.7. 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 4.7.1 (Regulärer Ausdruck)**.** *Sei Σ ein Alphabet. Die* regulären Ausdrücke *über Σ sind induktiv definiert durch*

- *$\emptyset$ ist ein regulärer Ausdruck.*
- *ε ist ein regulärer Ausdruck.*
- *a mit $a \in \Sigma$ ist ein regulärer Ausdruck.*
- *Wenn α_1 und α_2 reguläre Ausdrücke sind, dann ist auch $\alpha_1\alpha_2$ ein regulärer Ausdruck.*
- *Wenn α_1 und α_2 reguläre Ausdrücke sind, dann ist auch $(\alpha_1|\alpha_2)$ ein regulärer Ausdruck.*
- *Wenn α ein regulärer Ausdruck ist, dann auch $(\alpha)^*$.*

Die von einem regulären Ausdruck α erzeugte Sprache *$L(\alpha)$ ist rekursiv 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 $(\alpha_1|\alpha_2|\ldots|\alpha_n)$.

Beispiel 4.7.2. *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 4.7.3. *Alle endlichen Sprachen sind durch reguläre Ausdrücke beschreibbar: Sei $S = \{w_1, \ldots, w_n\}$ eine endliche Sprache, dann erzeugt $(w_1|w_2|\ldots|w_n)$ die Sprache S.*

Theorem 4.7.4 (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.

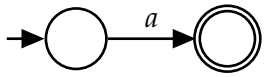
Wir beweisen beide Teile unabhängig voneinander.

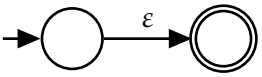
1. Sei α ein regulärer Ausdruck. Wir konstruieren per Induktion über die Größe von α einen NFA M_α mit ε-Übergängen und eindeutigen Start- und Endzuständen, der $L(\alpha)$ akzeptiert. Die Größe $|\alpha|$ eines regulären Ausdrucks α ist die Anzahl der Operatoren in dem Ausdruck, also $|\varepsilon| = |\emptyset| = 1$; $|a| = 1$ für alle $a \in \Sigma$; $|\alpha_1\alpha_2| = |(\alpha_1 \mid \alpha_2)| = |\alpha_1| + |\alpha_2| + 1$; $|\alpha^*| = |\alpha| + 1$. (Konkatenation und Einbettung von Symbolen $a \in \Sigma$ zählen auch als Operatoren, obwohl unsere Notation keine expliziten Zeichen für diese Operationen vorsieht.)

Als Induktionsbasis betrachten wir Ausdrücke α mit $|\alpha| = 0$. Nach Definition von $|\alpha|$ gibt es keine solchen Ausdrücke, also ist nichts zu zeigen.

Im Induktionsschritt betrachten wir Ausdrücken α mit $|\alpha| = n$ und $n \in \mathbb{N}_{>0}$. Als Induktionshypothese nehmen wir an, dass es für jeden Ausdruck α' mit $|\alpha'| < |\alpha|$ einen NFA $M_{\alpha'}$ mit den oben genannten Eigenschaften gibt, der $L(\alpha')$ akzeptiert.

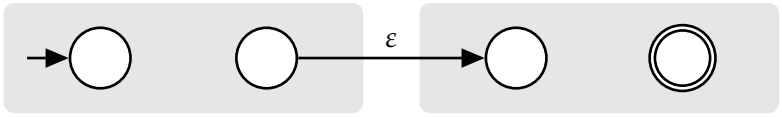
Vollständige Fallunterscheidung über die Form von α:

- Falls $\alpha = a \in \Sigma$ konstruiere den Automaten 

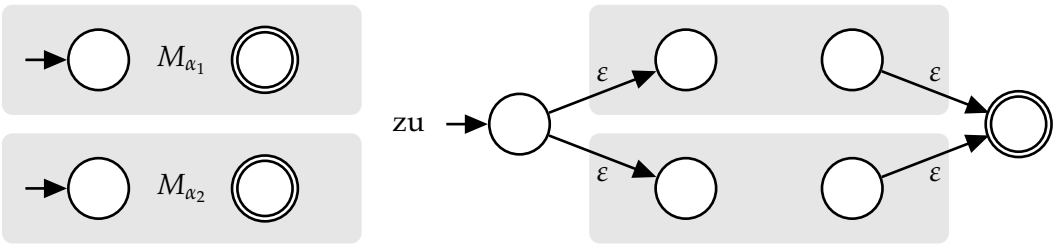
.
- Falls $\alpha = \varepsilon$ konstruiere den Automaten 

.
- Falls $\alpha = \emptyset$ konstruiere den Automaten
.
- Falls $\alpha = \alpha_1\alpha_2$, so liefert die Induktionshypothese NFAs mit ε-Übergängen und eindeutigen Start- und Endzuständen M_{α_i} mit $L(\alpha_i) = L(M_{\alpha_i})$ für $i = 1, 2$. Konstruiere M_α wie folgt: Verbinde den Endzustand von M_{α_1} mit einem ε-Übergang mit dem Startzustand von M_{α_2} und mache den Startzustand von M_{α_1} zum neuen Startzustand von M_α und den Endzustand von M_{α_2} zum neuen Endzustand von M_α. Bildlich skizziert wird

  zu

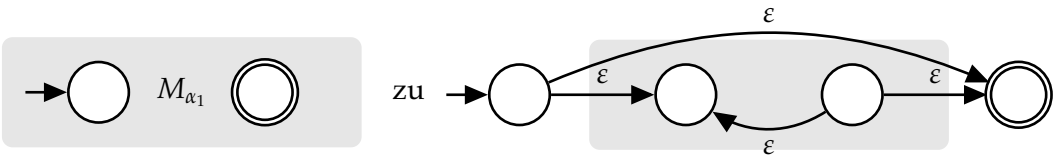
  

  Offensichtlich gilt $L(M_\alpha) = L(M_{\alpha_1})L(M_{\alpha_2})$. Mit der Induktionsannahme gilt daher $L(M_\alpha) = L(\alpha_1)L(\alpha_2)$. Definition 4.7.1 liefert $L(\alpha_1\alpha_2) = L(\alpha_1)L(\alpha_2)$ und daher $L(M_\alpha) = L(\alpha)$.
- Falls $\alpha = (\alpha_1|\alpha_2)$, so liefert die Induktionshypothese NFAs $M_{\alpha_1}, M_{\alpha_2}$ mit ε-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 ε-Übergang von z_0 zum Startzustand von M_{α_i} hinzu. Für $i = 1, 2$ füge einen ε-Übergang vom Endzustand von M_{α_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 4.7.1 liefert $L(\alpha_1|\alpha_2) = L(\alpha_1) \cup L(\alpha_2)$ und daher $L(M_\alpha) = L(\alpha)$.

- Falls $\alpha = (\alpha_1)^*$, so liefert die Induktionshypothese einen NFA mit ε-Übergängen und eindeutigem Start- und Endzustand M_{α_1} mit $L(M_{\alpha_1}) = L(M)$. Konstruiere M_α durch Hinzufügen eines neuen Startzustands, eines neuen Endzustands und von ε-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 ε-Übergang vom neuen Start- zum neuen Endzustand wird eingefügt, um das 0-fache Wiederholen von α zu ermöglichen (und damit das leere Wort ε 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. Der neue Übergang vom alten Endzustand zum alten Startzustand kann genommen werden, jedes mal 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 4.7.1 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 α sodass $L(M) = L(\alpha)$ gilt. O.B.d.A. sei $Z = \{z_1, \ldots, z_n\}$. Für $w \in \Sigma^*$ und $z_i \in Z$ mit $\widetilde{\delta}(z_i, w) = z_j$ sei $visit_i(w) = q_1, \ldots, q_m$ die Folge der durch den DFA besuchten Zustände (wobei $q_1 = z_i$ und $q_m = z_j$).

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

$$L^k_{i,j} = \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} \widetilde{\delta}(z_i, w) = z_j \text{ und } visit_i(w) = q_1, \ldots, q_m, \\ \text{sodass für } l = 2, \ldots, m-1\text{: wenn } q_l = z_p \text{ dann } p \leq k \end{array} \right\}$$

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

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

   Für den Induktionsschritt verwenden wir als Induktionsannahme, dass es für alle i, j und

festem k einen regulären Ausdruck $\alpha^k_{i,j}$ gibt, der $L^k_{i,j}$ erzeugt. Es gilt

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

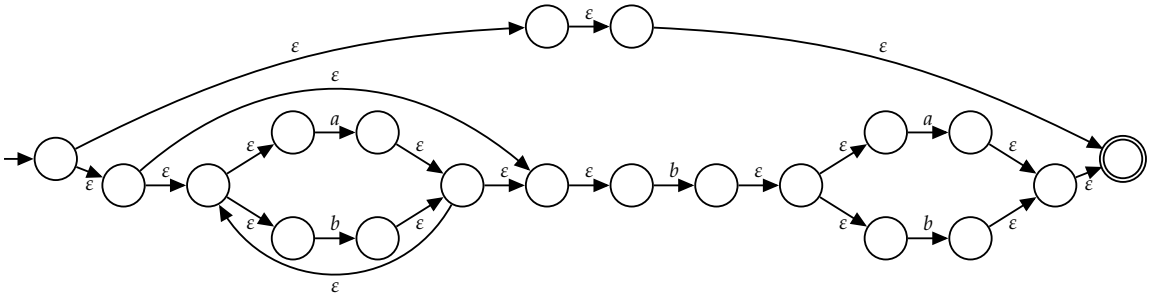
denn ein Lauf von z_i zu z_j kann entweder den Zustand z_{k+1} nicht als Zwischenzustand verwenden (dieser Fall wird durch $L^k_{i,j}$ abgedeckt) oder der Lauf kann aufgespalten werden in drei Teile die sequentiell aufeinander folgen:

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

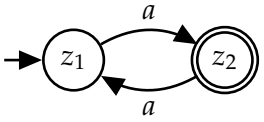
Der reguläre Ausdruck, der $L^{k+1}_{i,j}$ erzeugt, ist daher $\alpha^{k+1}_{i,j} = (\alpha^k_{i,j} | \alpha^k_{i,k+1}(\alpha^k_{k+1,k+1})^* \alpha^k_{k+1,j})$. Die darin verwendeten regulären Ausdrücke sind alle von der Form $\alpha^k_{l,l'}$ und daher durch die Induktionsannahme verfügbar.

Schließlich können wir den regulären Ausdruck angeben, der $L(M)$ erzeugt: Sei $E = \{z_{i_1}, \ldots, z_{i_r}\}$. Wenn $r = 0$, dann gilt, dass $\emptyset$ die Sprache $L(M)$ erzeugt. Sonst gilt, dass $(\alpha^n_{1,i_1} | \alpha^n_{1,i_2} | \ldots | \alpha^n_{1,i_r})$ die Sprache $\bigcup_{z_i \in E} L^n_{1,i} = L(M)$ erzeugt. □

Beispiel 4.7.5. *Wir konstruieren zum regulären Ausdruck $(\varepsilon|(a|b)^*b(a|b))$ einen NFA mit ε-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 4.7.4).*



Beispiel 4.7.6. *Betrachte den folgenden DFA M*



dann ist der entsprechend Theorem 4.7.4 konstruierte reguläre Ausdruck

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

denn

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

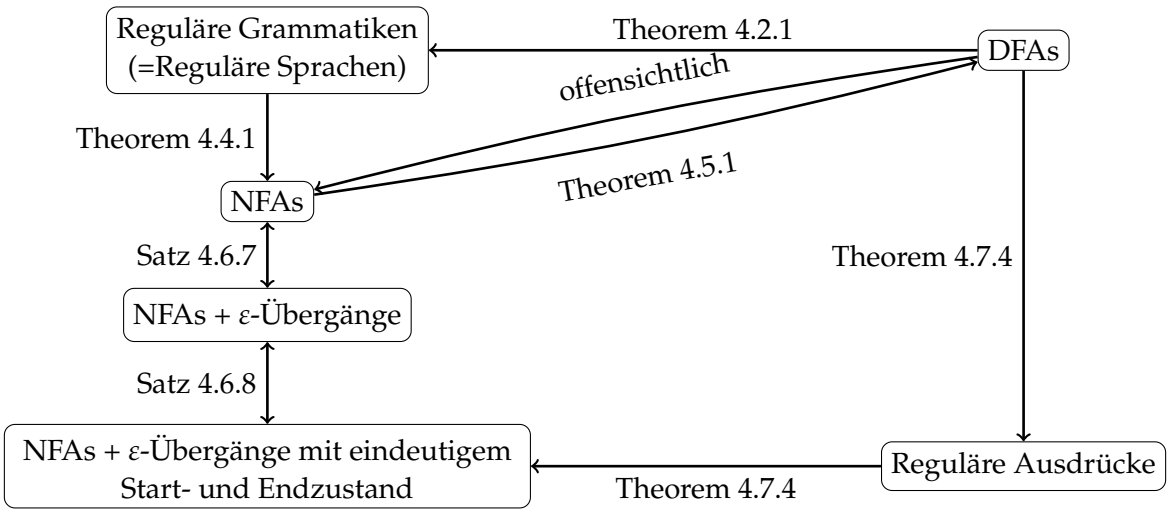
Für alle regulären Ausdrücke α gelten Gleichheiten wie $L\left((\varepsilon)^*\right) = L(\varepsilon)$, $L(\alpha\varepsilon) = L(\varepsilon\alpha) = L(\alpha)$, $L\left((\varepsilon|\alpha)^*\right) = L\left((\alpha)^*\right)$, $L\left((\varepsilon|\alpha)(\alpha)^*\right) = L\left((\alpha)^*(\varepsilon|\alpha)\right) = L\left((\alpha)^*\right)$ und $L(\alpha|\alpha) = L(\alpha)$. Mithilfe von solchen Gleichheiten kann man den regulären Ausdruck $\big((a|\varepsilon(\varepsilon)^*a)|(a|\varepsilon(\varepsilon)^*a)(\varepsilon|a(\varepsilon)^*a)^*(\varepsilon|a(\varepsilon)^*a)\big)$ zu $a(aa)^*$ vereinfachen. Die repräsentierte Sprache besteht aus allen Wörtern a^i mit $i \in \mathbb{N}_{>0}$ und i ungerade.

4.8. Zusammenfassende Darstellung der Formalismen für reguläre Sprachen

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

Das folgende Schaubild zeigt nochmal die Zusammenhänge und gibt Referenzen auf die gezeigten Sätze und Theoreme. Ein Pfeil $A \to B$ meint hierbei, dass wir eine Kodierung des einen Formalismus A in den anderen Formalismus B angegeben haben.

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



4.9. 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).

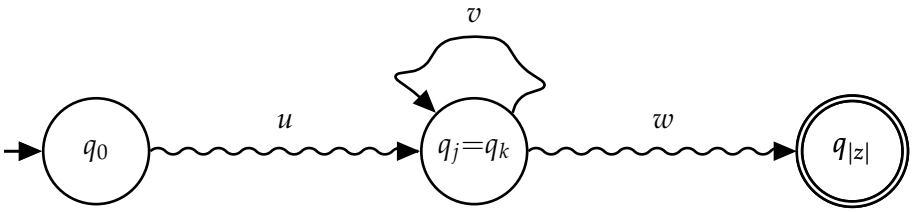
Die Idee der Pumping-Eigenschaft geht aus der Beobachtung hervor, dass ein DFA eine *endliche* Menge von Zuständen besitzt, und daher – sofern er eine unendliche Menge von Wörtern akzeptiert – Zustände mehrfach besuchen muss. Tatsächlich kann man sich klarmachen, dass

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 4.9.1 (Pumping-Lemma)**.** *Sei L eine reguläre Sprache. Dann gilt die folgende, sogenannte* Pumping-Eigenschaft *für L: Es gibt eine Zahl $n \in \mathbb{N}_{>0}$, sodass jedes Wort $z \in L$, das Mindestlänge n hat (d.h. $|z| \geq n$), als $z = uvw$ geschrieben werden kann, sodass gilt:*

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

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 $q_0, q_1, \ldots q_{|z|}$, 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 bezüglich dieser Eigenschaft). Sei u das Präfix von z, der von q_0 zu q_j führte (d.h. $\widetilde{\delta}(q_0, u) = q_j$), v das Teilwort, das von q_j zu q_k führte (d.h. $\widetilde{\delta}(q_j, v) = q_k$) und w das verbleibende Suffix mit $z = uvw$ für den $\widetilde{\delta}(q_k, w) = q_{|z|}$ 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 $\widetilde{\delta}(q_0, u) = q_j = \widetilde{\delta}(q_0, uv) = q_k$ und somit $\widetilde{\delta}(q_0, uw) = \widetilde{\delta}(q_0, uvw) = q_{|z|} \in E$, d.h. $uv^0w \in L(M)$. Sei $i > 0$, dann folgt aus $\widetilde{\delta}(q_j, v) = q_k = q_j$ auch $\widetilde{\delta}(q_j, v^i) = q_j$ und daher auch $\widetilde{\delta}(q_0, uv^iw) = \widetilde{\delta}(q_k, v^iw) = \widetilde{\delta}(q_j, w) = q_{|z|} \in E$. Daher gilt $uv^iw \in L(M)$ für alle $i \in \mathbb{N}$. □

Bemerkung 4.9.2. *Sei $L = \{w_1, \ldots, w_m\}$ eine endliche Sprache. Dann erfüllt L die Pumping-Eigenschaft, denn mit $k = \max\{|w_i| \mid i = \{1, \ldots, m\}\}$ (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

$$L \text{ ist regulär} \implies L \text{ erfüllt die Pumping-Eigenschaft}$$

ist äquivalent zur Aussage

L erfüllt die Eigenschaften des Pumping-Lemma *nicht* $\implies$ L ist *nicht* regulär

Satz 4.9.3. *Die Sprache $L = \{a^j b^j \mid j \in \mathbb{N}\}$ ist nicht regulär.*

Beweis. Um Nichtregularität zu zeigen, reicht es durch das Pumping-Lemma (Lemma 4.9.1) zu zeigen, dass L die Pumping-Eigenschaft nicht hat. Der Beweis ist durch Widerspruch: Wir nehmen an, dass L die Pumping-Eigenschaft hat und leiten einen Widerspruch her.

Sei $n \in \mathbb{N}_{>0}$ beliebig. Wir wählen für $z \in L$ das Wort $a^n b^n$ (für welches $|z| \geq n$ erfüllt ist). Sei $z = uvw$ eine beliebige Zerlegung von z, sodass $|uv| \leq n$, $|v| \geq 1$ und $uv^i w \in L$ für alle $i \in \mathbb{N}$. Dann gilt $u = a^r, v = a^{n-r}$ mit $r < n$. Daher können wir z.B. $i = 2$ wählen 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 hätten z.B. auch $i = 0$ wählen können.) Nach Annahme gilt aber $uv^i w \in L$ für jedes $i \in \mathbb{N}$, insbesondere für $i = 2$. Widerspruch. □

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 $n \in \mathbb{N}_{>0}$.
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 4.9.4. *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 $n \in \mathbb{N}_{>0}$ 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. □

Satz 4.9.5. *Die Sprache $L = \{a^n \mid n \text{ ist eine Quadratzahl}\}$ ist nicht regulär.*

Beweis. Sei $n \in \mathbb{N}_{>0}$ 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. □

Satz 4.9.6. *Die Sprache $L = \{a^{2^n} \mid n \in \mathbb{N}\}$ ist nicht regulär.*

Beweis. Sei $n \in \mathbb{N}_{>0}$ 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 $uv^2w = a^{2^n+k}$ 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. □

Satz 4.9.7. *Die Sprache $L = \{w \in \{a,b\}^* \mid w \text{ ist ein Palindrom}\}$ ist nicht regulär.*

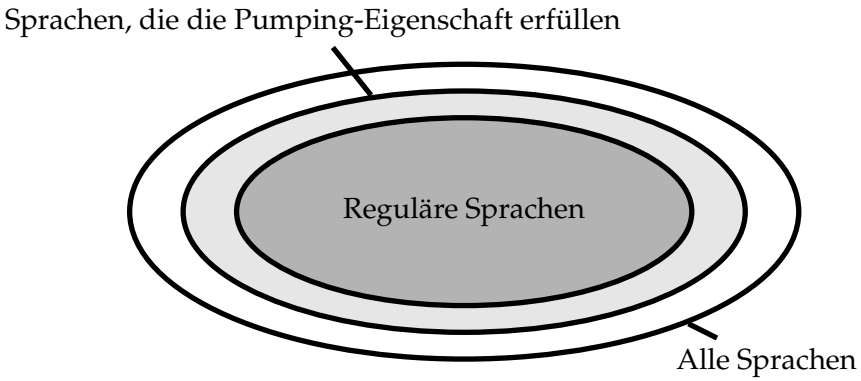
Beweis. Sei $n \in \mathbb{N}_{>0}$ beliebig. Wir wählen $z = a^n b a^n$ als Wort mit Mindestlänge n. Sei $z = uvw$ mit $|uv| \leq n$ und $|v| \geq 1$. Dann ist $uv^0w = a^k b a^n$ mit $k = n - |v| < n$ kein Palindrom. Mit dem Pumping-Lemma folgt, dass L nicht regulär ist. □

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

Lemma 4.9.8. *Es gibt Sprachen, welche die Pumping-Eigenschaft (siehe Lemma 4.9.1) erfüllen aber nicht regulär sind. Die Sprache $L = \{a^k b^l c^l \mid k,l \in \mathbb{N}\} \cup \{b,c\}^*$ ist eine solche Sprache.*

Beweis. Wir zeigen hier nur, dass L die Pumping-Eigenschaft erfüllt (die Nichtregularität von L zeigen wir erst später in Satz 4.10.5). 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 $uv^iw \in \{b,c\}^* \subseteq L$ für alle $i \in \mathbb{N}$. 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 $uv^iw = a^{k+i-1} b^l c^l \in L$ für alle $i \in \mathbb{N}$, erfüllt L die Pumping-Eigenschaft. □

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 (endlicher Index der Nerode-Relation) betrachten wir im nächsten Abschnitt.

4.10. ⋆ Der Satz von Myhill und Nerode

Wir definieren die sogenannte Nerode-Relation (benannt nach Anil Nerode):

Definition 4.10.1 (Nerode-Relation $\sim_L$)**.** *Sei L eine formale Sprache über Σ. Die* Nerode-Relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$ *zu L ist definiert für alle Wörter $u, v \in \Sigma^*$ 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 dasselbe Suffix.

Satz 4.10.2. *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 $u_1 w \in L$, dann folgt aus $u_1 \sim_L u_2$ auch $u_2 w \in L$, woraus mit $u_2 \sim_L u_3$ wiederum $u_3 w \in L$ folgt. Analog kann gezeigt werden, dass $u_3 w \in L$ auch $u_1 w \in L$ impliziert. Daher gilt $u_1 w \in L \iff u_3 w \in L$ und damit $u_1 \sim_L u_3$. □

Wir erinnern, dass eine Äquivalenzrelation die Grundmenge Σ^* in disjunkte Äquivalenzklassen $[u_1]_{\sim_L}, [u_2]_{\sim_L}, \ldots$ zerlegt und dass der Index einer Äquivalenzrelation die Anzahl der disjunkten Äquivalenzklassen ist (u.U. ∞ bei unendlich vielen Äquivalenzklassen). John Myhill und Anil Nerode bewiesen in den 1950er Jahren den folgenden Satz:

Theorem 4.10.3 (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 $u \approx_M v \iff \widetilde{\delta}(z_0, u) = \widetilde{\delta}(z_0, v)$, 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 $\widetilde{\delta}(z_0, uw) = \widetilde{\delta}(\widetilde{\delta}(z_0, u), w) = \widetilde{\delta}(\widetilde{\delta}(z_0, v), w) = \widetilde{\delta}(z_0, vw)$ 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 $n \in \mathbb{N}_{>0}$ und Repräsentanten $u_1, \ldots, u_n$, sodass $\Sigma^* = [u_1]_{\sim_L} \cup \ldots \cup [u_n]_{\sim_L}$. Wir definieren den sogenannten *Nerode-Automaten*:

   Sei $M = (Z, \Sigma, \delta, [\varepsilon]_{\sim_L}, E)$ ein DFA mit $Z = \{[u_1]_{\sim_L}, \ldots, [u_n]_{\sim_L}\}$, $\delta([u_i]_{\sim_L}, a) = [u_i a]_{\sim_L}$ für alle $a \in \Sigma$ und $E = \{[u_i]_{\sim_L} \mid i \in \{1, \ldots, n\}, u_i \in L\}$. Wir zeigen $w \in L(M) \iff w \in L$:

$w \in L(M)$ ist äquivalent zu $\widetilde{\delta}([\varepsilon]_{\sim_L}, w) \in E$, was (anhand der Definition von δ) 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. □

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 $i \in \mathbb{N}$, 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 4.10.4. *Betrachte die Sprache $L = \{a^n b^n \mid n \in \mathbb{N}_{>0}\}$ und deren Nerode-Relation $\sim_L$. Die Äquivalenzklassen $[a^i b]_{\sim_L}$ für $i \in \mathbb{N}_{>0}$ enthalten jeweils alle Wörter, denen noch $i-1$ b's fehlen, um in L enthalten zu sein. Z.B. ist $[ab]_{\sim_L} = L$, $[a^2 b]_{\sim_L} = \{a^2 b, a^3 b^2, a^4 b^3, \ldots\}$, $[a^3 b]_{\sim_L} = \{a^3 b, a^4 b^2, a^5 b^3, a^6 b^4, \ldots\}$. 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 4.10.3) zeigt, dass L nicht regulär sein (was wir bereits wussten, durch Beweis mit dem Pumping-Lemma).*

Bisher konnten wir nicht zeigen, dass die Sprache $L = \{a^k b^l c^l \mid k, l \in \mathbb{N}_{>0}\}$ nicht regulär ist, da das Pumping-Lemma nicht anwendbar ist (siehe Lemma 4.9.8). Mithilfe des Satzes von Myhill und Nerode holen wir das jetzt nach:

Satz 4.10.5. *Die Sprache $L = \{a^k b^l c^l \mid k, l \in \mathbb{N}\} \cup \{b, c\}^*$ ist nicht regulär.*

Beweis. Betrachte die Äquivalenzklassen $[ab^i c]_{\sim_L}$ für $i \in \mathbb{N}_{>0}$. Diese sind alle paarweise verschieden, da für $i, j \in \mathbb{N}_{>0}$ 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 ∞ und Theorem 4.10.3 zeigt daher, dass L nicht regulär ist. □

Beispiel 4.10.6. *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:*

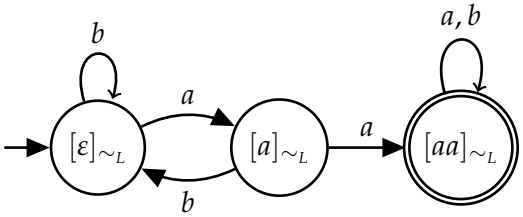
$$\begin{aligned}
[\varepsilon]_{\sim_L} &= \{w \in \Sigma^* \mid w \text{ enthält keine zwei aufeinander folgende a und endet nicht mit a}\} \\
[a]_{\sim_L} &= \{w\Sigma^* \mid w \text{ enthält keine zwei aufeinander a und endet mit a}\} \\
[aa]_{\sim_L} &= \{w \in \Sigma^* \mid w \text{ enthält zwei aufeinanderfolgende a}\} = L
\end{aligned}$$

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 Wörter in den Mengen äquivalent sind und die Beschreibungen der Mengen stimmen. ($[\varepsilon]_{\sim_L}$ sind alle Wörter u denen noch ein Wort w aus L angehängt werden muss, damit $uw \in L$ gilt, $[a]_{\sim_L}$ sind alle Wörter 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 Wörter u, für die für alle $w \in \Sigma^*$ gilt $uw \in L$). Die drei Äquivalenzklassen sind paarweise disjunkt: Für ε 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 ε 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-Automaten), wie er im Beweis von Theorem 4.10.3 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

$$\begin{array}{lll}
\delta([\varepsilon]_{\sim_L}, a) = [a]_{\sim_L} & \delta([a]_{\sim_L}, a) = [aa]_{\sim_L} & \delta([aa]_{\sim_L}, a) = [aa]_{\sim_L} \\
\delta([\varepsilon]_{\sim_L}, b) = [\varepsilon]_{\sim_L} & \delta([a]_{\sim_L}, b) = [\varepsilon]_{\sim_L} & \delta([aa]_{\sim_L}, b) = [aa]_{\sim_L}
\end{array}$$

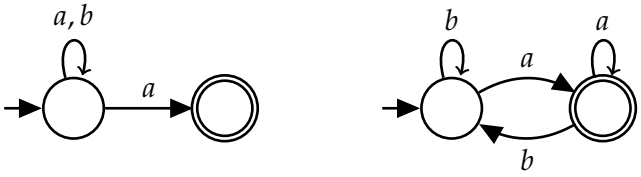
Der Zustandsgraph von M ist



Satz 4.10.7. *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 4.10.3 zeigt, dass $\approx_{M'}$ eine Verfeinerung von $\sim_L$ ist ($\approx_{M'} \subseteq \sim_L$). Zudem gilt $Index(\approx_M) = Index(\sim_L)$. Daher ist $Index(\approx_{M'}) \geq Index(\approx_M)$ und daher auch $|Z'| \geq Index(\approx_{M'}) \geq Index(\approx_M) = |Z|$. (Für den Nerode-Automaten gilt $Index(\approx_M) = |Z|$. Für einen beliebigen Automaten M' gilt nur $|Z'| \geq Index(\approx_{M'})$, wegen unerreichbaren Zuständen.)
Sei nun $|Z'| = |Z|$, d.h. M' hat 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 $f : Z \to Z'$ mit $f([u]_{\approx_M}) = f([u]_{\sim_L}) := [u]_{\approx_M'}$ eine Umbenennung der Zustände 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. □

Die letzte Aussage zeigt, dass alle minimalen DFAs bis auf Umbenennung von Zuständen identisch sind. Beachte, dass dies für NFAs im Allgemeinen nicht richtig ist. Es gibt strukturell unterschiedliche NFAs mit derselben 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:



4.11. Minimierung von Automaten

Dieser Abschnitt richtet sich teilweise nach (Weg99).

Definition 4.11.1 (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 $w \in \Sigma^* : \widetilde{\delta}(z, w) \in E \iff \widetilde{\delta}(z', w) \in E$. 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: $\delta([z]_\equiv, a)$ 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 $\widetilde{\delta}(z,w) \in E \iff \widetilde{\delta}(z',w) \in E$ und damit auch $\widetilde{\delta}(z,aw) \in E \iff \widetilde{\delta}(z',aw) \in E$ und damit auch $\widetilde{\delta}(\delta(z,a),w) \in E \iff \widetilde{\delta}(\delta(z',a),w) \in E$. D.h. es gilt dann auch $\delta(z,a) \equiv \delta(z',a)$.

Satz 4.11.2. *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.

1. Sei $w \in \Sigma^*$. Dann gilt:
   - M durchläuft die Zustandsfolge $q_0, \ldots, q_{|w|}$ entlang w und akzeptiert w g.d.w. $q_{|w|} \in E$ gilt.
   - M' durchläuft die Zustandsfolge $[q_0]_\equiv, \ldots, [q_{|w|}]_\equiv$ und akzeptiert w g.d.w. $[q_{|w|}]_\equiv \in E'$ 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.
2. ($\star$) Sei $L = L(M') = L(M)$. Da alle Zustände in M erreichbar sind, sind per Konstruktion auch alle Zustände in M' erreichbar. Für die Minimalität genügt es zu zeigen, dass M' nicht mehr als $\mathit{Index}(\sim_L)$ Zustände hat. Wir zeigen, dass es für jede Äquivalenzklasse $[v]_{\sim_L}$ nur maximal einen Zustand in M' gibt, der vom Startzustand entlang jedes Wortes $u \in [v]_{\sim_L}$ erreicht wird. Daraus folgt dann sofort (da alle Zustände in M' erreichbar sind), dass M' nicht mehr als $\mathit{Index}(\sim_L)$ Zustände hat. Seien $u, u' \in [v]_{\sim_L}$, d.h. $u \sim_L u'$. Dann gilt $\forall w \in \Sigma^* : uw \in L \iff u'w \in L$, woraus folgt $\forall w \in \Sigma^* : \widetilde{\delta'}(z_0', uw) \in E' \iff \widetilde{\delta'}(z_0', u'w) \in E'$, was weiter umgeformt werden kann zu $\forall w \in \Sigma^* : \widetilde{\delta'}(\widetilde{\delta'}(z_0', u), w) \in E' \iff \widetilde{\delta'}(\widetilde{\delta'}(z_0', u'), w) \in E'$, was äquivalent zu $\widetilde{\delta'}(z_0', u) = \widetilde{\delta'}(z_0', u')$ ist. □

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:

- Bilde eine Partition von Zuständen, die verschieden sein müssen. Eine *Partition* einer Menge A ist eine Menge von nichtleeren Mengen $\{A_1, \ldots, A_n\}$, sodass $A = A_1 \dot\cup \cdots \dot\cup A_n$. Die Mengen A_i heißen *Klassen* der Partition. Initial wissen wir, dass die Zustände in E und die in $Z \setminus E$ verschiedene Klassen sind.
- Verfeinere die Partition durch Untersuchen von Übergängen: Zustände, die über δ mit einem beliebigen Symbol $a \in \Sigma$ zu verschiedenen Klassen führen, müssen getrennt werden. Dieser Vorgang wird solange wiederholt bis sich nichts mehr ändert.
- Die Elemente einer Klasse stellen äquivalente Zustände dar.

In Algorithmus 3 sind diese Schritte in Pseudocode ausformuliert.

Satz 4.11.3. *Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA, der keine unerreichbaren Zustände hat. Algorithmus 3 für M berechnet die Äquivalenzklassen bezüglich $\equiv$.*

Algorithmus 3 : Berechnung aller äquivalenten Zustände

Eingabe : DFA $M = (Z, \Sigma, \delta, z_0, E)$, der keine unerreichbaren Zustände hat
Ausgabe : Partition $\mathcal{P} = \{[z_1]_{\equiv}, \ldots, [z_m]_{\equiv}\}$ von Z
Beginn
  initialisiere Partition $\mathcal{P}$ mit E (falls nicht leer) und $Z \setminus E$ (falls nicht leer) ;
  wiederhole
    für jedes $K = \{z_1, \ldots, z_m\} \in \mathcal{P}$ *mit* $|m| \geq 2$ **tue**
      für jedes $a \in \Sigma$ **tue**
        berechne die Partition $\mathcal{Q}$ von $\{z_1, \ldots, z_m\}$ über dem Wert von $[\delta(z_i, a)]$ für jedes i;
        $\mathcal{P} := (\mathcal{P} \setminus \{K\}) \cup \mathcal{Q}$;
      Ende
    Ende
  bis *sich* $\mathcal{P}$ *nicht mehr verändert*;
  return $\mathcal{P}$
Ende

Beweis. Wir führen den Beweis in zwei Schritten:

1. Wir müssen zeigen, dass wenn $z \equiv z'$, dann sind z und z' in derselben Klasse von $\mathcal{P}$.

   Wir zeigen die Kontraposition: Wenn z und z' in verschiedenen Klassen von $\mathcal{P}$ sind, dann $z \not\equiv z'$. Und um $z \not\equiv z'$ zu zeigen, reicht es, ein Wort w zu finden, sodass $\widetilde{\delta}(z, w) \in E$ und $\widetilde{\delta}(z', w) \notin E$ oder umgekehrt.

   Der Beweis ist durch Induktion über die Anzahl n der Schleifeniterationen bis z und z' getrennt wurden.

   - Fall $n = 0$: Da z und z' schon vor der ersten Iteration getrennt wurden, müssen $z \in E$ und $z' \notin E$ oder umgekehrt gelten. Wir nehmen $w = \varepsilon$: $\widetilde{\delta}(z, \varepsilon) \in E$ und $\widetilde{\delta}(z', \varepsilon) \notin E$ oder umgekehrt.
   - Fall $n > 0$: Da z und z' in Iteration n getrennt wurden, muss es $a \in \Sigma$ geben, sodass $\delta(z, a)$ und $\delta(z', a)$ schon in Iteration $n - 1$ getrennt waren. Die Induktionshypothese liefert ein Wort w' mit $\widetilde{\delta}(\delta(z, a), w') \in E$ und $\widetilde{\delta}(\delta(z', a), w') \notin E$ oder umgekehrt. Wir nehmen $w = aw'$: $\widetilde{\delta}(z, aw') \in E$ und $\widetilde{\delta}(z', aw') \notin E$ oder umgekehrt.

2. Wir müssen zeigen, dass wenn $z \not\equiv z'$, dann sind z und z' in zwei verschiedenen Klassen von $\mathcal{P}$.

   Sei w ein Wort sodass $\widetilde{\delta}(z, w) \in E$ und $\widetilde{\delta}(z', w) \notin E$ oder umgekehrt.

   Der Beweis ist durch Induktion über $|w|$.

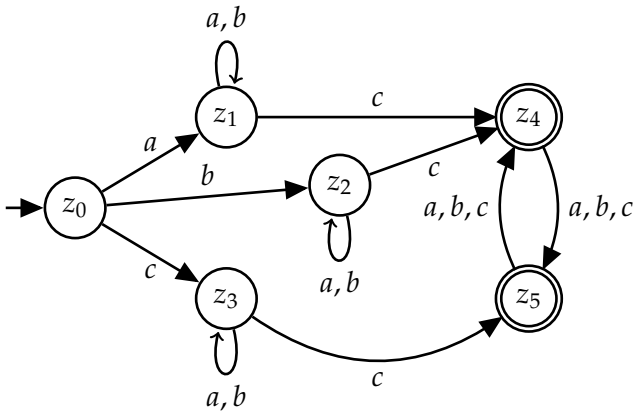
   - Fall $w = \varepsilon$: Dann haben wir $\widetilde{\delta}(z, \varepsilon) \in E$ und $\widetilde{\delta}(z', \varepsilon) \notin E$ oder umgekehrt. Die Initialisierungsphase sorgt dafür, dass z und z' getrennt werden.
   - Fall w ist von der Form aw': Dann haben wir $\widetilde{\delta}(z, aw') = \widetilde{\delta}(\delta(z, a), w') \in E$ und $\widetilde{\delta}(z', aw') = \widetilde{\delta}(\delta(z', a), w') \notin E$ oder umgekehrt. Per Induktionshypothese müssen $\delta(z, a)$ und $\delta(z', a)$ getrennt sein. Der Algorithmus muss z und z' trennen, wenn a betrachtet wird. □

In Algorithmus 4 wird er komplette Algorithmus zur Zustandsminimierung von DFAs beschrieben.

Algorithmus 4 : 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 3;
erzeuge den Äquivalenzklassenautomat, indem die berechneten äquivalenten Zustände vereinigt werden;
Ende

Beispiel 4.11.4. *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.

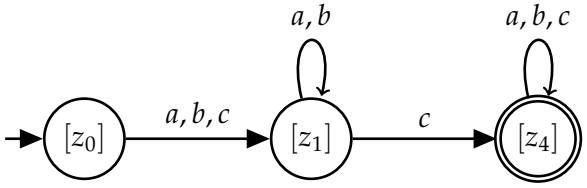
Die Partition $\mathcal{P}$ wird erstellt als

z_0 z_1 z_2 z_3	z_4 z_5

Wir betrachten das Symbol c. Der Zustand $\delta(z_0, c)$ befindet sich in der ersten Klasse ($\{z_0, z_1, z_2, z_3\}$), während $\delta(z_1, c)$, $\delta(z_2, c)$ und $\delta(z_3, c)$ sich in der zweiten Klasse ($\{z_4, z_5\}$) befinden. Daher müssen wir z_0 von z_1, z_2, z_3 trennen. Dies ergibt folgende Partition:

z_0	z_1 z_2 z_3	z_4 z_5

Egal, ob wir a, b oder c betrachten, verändert sich die Partition nicht mehr. Das ergibt 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



4.12. Abschlusseigenschaften der regulären Sprachen

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

Satz 4.12.1. *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 α_1, α_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 4.7.4). □

Satz 4.12.2. *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 $(\widetilde{\delta}(z_0, w) \in E) \iff \neg(\widetilde{\delta}(z_0, w) \in Z \setminus E)$ □

Satz 4.12.3. *Die regulären Sprachen sind abgeschlossen bezüglich Schnitt.*

Beweis. Das folgt aus den Sätzen 4.12.1 und 4.12.2 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: $\widetilde{\delta}((z_{0,1}, z_{0,2}), w) \in (E_1, E_2) \iff \left(\widetilde{\delta_1}(z_{0,1}, w) \in E_1 \wedge \widetilde{\delta_2}(z_{0,2}, w) \in E_2\right)$. □

Zusammenfassend halten wir fest:

Theorem 4.12.4 (Abschlusseigenschaften der regulären Sprachen)**.** *Die regulären Sprachen sind abgeschlossen bezüglich Vereingung, Schnitt, Komplementbildung, Produkt und Kleeneschem Abschluss.*

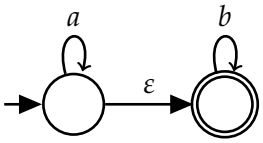
Beachte, dass die Abschlusseigenschaften der regulären Sprachen nicht nur nützlich sind, um aus regulären Sprachen weitere reguläre Sprachen zu konstruieren, sondern sie können auch verwendet werden, um die *Nichtregularität* von Sprachen zu zeigen.

Satz 4.12.5. *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 4.12.4), gilt dann auch $\overline{L} = \{a\}^* \setminus L = \{a^n \mid \text{n ist eine Primzahl}\}$ ist regulär. Das ist jedoch ein Widerspruch zu Satz 4.9.4, der zeigt, dass $\overline{L}$ nicht regulär ist. Daher war unsere Annahme falsch, L kann nicht regulär sein. □

Satz 4.12.6. *Die Sprache $L = \{w \in \{a,b\}^* \mid \#_a(w) = \#_b(w)\}$ ist nicht regulär.*

Beweis. Nehme an, L ist regulär. Die Sprache $L' = \{a^n b^m \mid n, m \in \mathbb{N}\}$ ist regulär, da z.B. der folgende NFA mit ε-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 $L \cap L' = \{a^j b^j \mid j \in \mathbb{N}\}$ ist dies ein Widerspruch zu Satz 4.9.3. Also war unsere Annahme falsch: L ist nicht regulär. □

4.13. Entscheidbarkeitsresultate zu regulären Sprachen

Wie bereits gezeigt, ist das Wortproblem für Typ 1,2,3-Grammatiken entscheidbar (siehe Satz 3.3.2 und Korollar 3.3.3). 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 $\widetilde{\delta}(z_0, w)$ braucht für einen DFA nur $|w|$ Schritte.

Wir betrachten nun die weiteren Probleme, wie wir sie in Abschnitt 3.4 definiert haben und zeigen, dass allesamt für die Klasse der regulären Sprachen entscheidbar sind.

Satz 4.13.1. *Das Leerheitsproblem für reguläre Sprachen ist entscheidbar.*

Beweis. Sei L eine reguläre Sprache, und M ein DFA mit $L(M) = L$. Dann gilt $L = \emptyset$ g.d.w. es keinen Pfad vom Startzustand zu einem Endzustand in M gibt. Dies kann man leicht mit einer Tiefensuche auf dem Zustandsgraph von M prüfen. □

Satz 4.13.2. *Das Endlichkeitsproblem für reguläre Sprachen ist entscheidbar.*

Beweis. Sei L eine reguläre Sprache und M ein DFA mit $L(M) = L$. Dann gilt $|L| < \infty$ g.d.w. 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 prüfen. □

Satz 4.13.3. *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 4.12.3) und prüfe anschließend das Leerheitsproblem für M. □

Satz 4.13.4. *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. □

5. 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 Klammerungen 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

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

geklammerte arithmetische Ausdrücke.

Die Sprache $L = \{a^j b^j \mid j \in \mathbb{N}\}$ ist nicht regulär (siehe Satz 4.9.3), aber kontextfrei, denn sie wird z.B. durch die Produktionen $S \to \varepsilon \mid T$ und $T \to aTb \mid ab$ und S als Startsymbol erzeugt.

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

5.1. ⋆ Einfache Operationen auf CFGs

In diesem Abschnitt beweisen wir einige Hilfssätze, die Operationen auf CFGs durchführen, ohne deren erzeugte Sprache zu ändern.

Lemma 5.1.1 (Inlining von Produktionen)**.** *Sei $G = (V, \Sigma, P, S)$ eine CFG, sei $A \to uBv \in P$, seien $B \to w_1 \mid \dots \mid w_n$ alle Regeln mit B als linker Seite und sei*

$$G' = (V, \Sigma, P \setminus \{A \to uBv\} \cup \{A \to uw_1v \mid uw_2v \mid \dots \mid uw_nv\}, S)$$

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 Ableitungs-

schritte, die $A \to uBv$ und anschließendes $B \to w_i$ im Syntaxbaum anwenden, durch einen Schritt $A \to uw_iv$ 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. □

Lemma 5.1.2 (Boxing von Satzformen). *Sei G eine CFG mit $G = (V, \Sigma, P \cup \{A \to w_1 \cdots w_n\}, S)$. Seien $B_1, \ldots, B_n$ neue unterschiedliche Variablen (d.h. $V \cap \{B_1, \ldots, B_n\} = \emptyset$ und $B_i \neq B_j$ wenn $i \neq j$) und sei $G' = (V \cup \{B_1, \ldots, B_n\}, \Sigma, P \cup \{A \to B_1 \ldots B_n, B_1 \to w_1, \ldots, B_n \to w_n\}, S)$.*
Dann gilt $L(G') = L(G)$.

Beweis. Jeder Ableitungsschritt $uAv \Rightarrow_G uw_1 \cdots w_nv$ kann „übersetzt“ werden in $uAv \Rightarrow_{G'} uB_1 \cdots B_nv \Rightarrow^n_{G'} uw_1 \cdots w_nv$. Umgekehrt können in jeder Ableitung $S \Rightarrow^*_{G'} w$, die Anwendungen der Regel $A \to B_1 \cdots B_n$, $B_i \to w_i$ 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. □

Eine Produktion nennt man *links-rekursiv*, wenn sie von der Form $A \to Au$ ist, und *rechts-rekursiv*, wenn sie von der Form $A \to uA$ 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 5.1.3 (Elimination der Links-Rekursion). *Sei $G = (V, \Sigma, P, S)$ eine CFG mit $P = P' \cup \{A \to Au_1 \mid \cdots \mid Au_n \mid w_1 \mid \cdots \mid w_m\}$, wobei $A \to Au_1 \mid \cdots \mid Au_n \mid w_1 \mid \cdots \mid w_m$ alle Produktionen in P mit Variable A als linker Seite sind, und die Satzformen $w_1, \ldots, w_m$ alle nicht mit A beginnen. Sei B eine neue Variable. Für die CFG $G' = (V \cup \{B\}, \Sigma, P'', S)$ mit*

$$P'' = P' \cup \{A \to w_1B \mid \cdots \mid w_mB \mid w_1 \mid \cdots \mid w_m,\ \ B \to u_1 \mid \cdots \mid u_n \mid u_1B \mid \cdots \mid u_nB\}$$

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 $A \Rightarrow^*_G w_r\, u_{f(1)} \cdots u_{f(k)}$ für eine Funktion $f : \{1, \ldots, k\} \to \{1, \ldots, n\}$ (mit $k \geq 0$) und ein $r \in \{1, \ldots, m\}$. Für G' können wir ebenfalls herleiten $A \Rightarrow^*_{G'} w_ru_{f(1)} \cdots u_{f(k)}$, wobei kein B mehr in $w_ru_{f(1)} \cdots u_{f(k)}$ vorkommt. Sei nun $w \in L(G)$ und $S \Rightarrow^*_G w$ eine Linksableitung von w mit Grammatik G. Ersetze alle Ableitungssequenzen von $x_1Ax_2 \Rightarrow^*_G x_1w_r\, u_{f(1)} \cdots u_{f(k)}x_2$ durch $x_1Ax_2 \Rightarrow^*_{G'} x_1w_ru_{f(1)} \cdots u_{f(k)}x_2$. 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 $A \Rightarrow^*_{G'} w_r\, u_{f(1)} \cdots u_{f(k)}$ für eine Funktion $f : \{1, \ldots, k\} \to \{1, \ldots, n\}$ (mit $k \geq 0$) und ein $r \in \{1, \ldots, m\}$. Dann gibt es (eine Linksableitung) $A \Rightarrow^*_G w_r\, u_{f(1)} \cdots u_{f(k)}$. Sei nun $w \in L(G')$ und $S \Rightarrow^*_{G'} w$ eine Rechtsableitung von w mit Grammatik G'. Ersetze alle Ableitungssequenzen von $x_1Ax_2 \Rightarrow^*_{G'} x_1w_r\, u_{f(1)} \cdots u_{f(k)}x_2$ durch $x_1Ax_2 \Rightarrow^*_G x_1w_ru_{f(1)} \cdots u_{f(k)}x_2$. Anschließend sind alle $\Rightarrow_{G'}$-Ableitungsschritte auch $\Rightarrow_G$-Ableitungsschritte und daher gibt es eine Ableitung $S \Rightarrow^*_G w$. □

5.2. Chomsky-Normalform

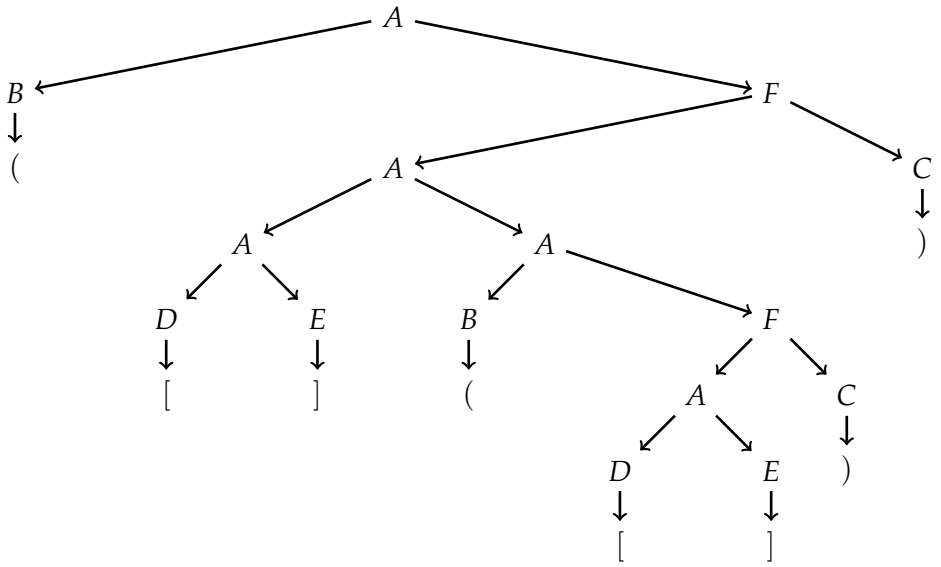
Die Chomsky-Normalform fordert eine spezielle Form aller Produktionen. Solche Normalformen sind beispielsweise nützlich, um Algorithmen auf Grammatiken zu formulieren und zu analysieren, da man sich auf die spezielle Form der Produktionen einschränken kann und daher weit weniger Fälle zu betrachten hat,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

Definition 5.2.1. *Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ mit $\varepsilon \notin L(G)$ ist in* Chomsky-Normalform*, wenn für $A \to w \in P$ gilt: $w = a \in \Sigma$ oder $w = BC$ mit $B, C \in V$.*

Beispiel 5.2.2. *Die Grammatik $G = (\{A\}, \{(,), [,]\}, \{A \to (A) \mid () \mid [A] \mid [] \mid A\,A\}, A)$ ist* nicht *in Chomsky-Normalform (nur die Produktion $A \to A\,A$ passt zum vorgeschriebenen Format). Hingegen ist $G' = (\{A, B, C, D, E, F, G\}, \{(,), [,]\}, \{A \to BF \mid BC \mid DG \mid DE \mid AA, B \to (, C \to), D \to [, E \to], F \to AC, G \to AE\}, A)$ in Chomsky-Normalform (und erzeugt dieselbe 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 5.2.3. *Eine Linksableitung des (8-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:*



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 1 gemacht, als wir ε-Produktionen entfernten.

5.2.1. ⋆ Entfernen von Einheitsproduktionen

Im nächsten Schritt möchten wir Produktionen der Form $A \to B$ 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 $A \to w$ zur Grammatik hinzufügen. Allerdings muss man dieses Ersetzen in der richtigen Reihenfolge tun (wenn w selbst eine Variable C ist, ist das Hinzufügen von $A \to C$ erst einmal nicht zielführend) und Sonderfälle (z.B. $A \to B$ und $B \to A$) anders behandeln. Der Algorithmus 5 leistet dies.

Algorithmus 5 : 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 $A \to B \in P$ gibt;
  solange *es einen Zyklus* $(A_1, A_2), \ldots, (A_{n-1}, A_n), (A_n, A_1) \in E$ *gibt* **tue**
    $P := P \setminus \{A_1 \to A_2, \ldots, A_{n-1} \to A_n, A_n \to A_1\}$; /* entferne die zyklischen Regeln */
    $P := P[A_1/A_2, \ldots, A_1/A_n]$; /* ersetze alle Vorkommen von A_i durch A_1 für $i = 2, \ldots, n$ */
    $V := V \setminus \{A_2, A_3, \ldots, A_n\}$; /* lösche $A_2, \ldots, A_n$ */
    $S := S[A_1/A_2, \ldots, A_1/A_n]$; /* ersetze Startsymbol durch A_1, falls es A_i, $2 \leq i \leq n$ war */
    $E := E \setminus \{(A_1, A_2), \ldots, (A_{n-1}, A_n), (A_n, A_1)\}$; /* entferne Zyklus aus Graph */
    $E := E[A_1/A_2, \ldots, A_1/A_n]$; /* ersetze alle A_i durch A_1 für $i = 2, \ldots, n$ in den Kanten */
  Ende
  Sortiere D topologisch und nummeriere die Variablen in V durch (und benenne entsprechend in E, P, S um), sodass gilt: $A_i \to A_j$ impliziert $i < j$;
  Sei $V = \{A_1, \ldots, A_k\}$;
  für $i = k$ *bis* 1 **tue**
    wenn $A_i \to A_j \in P$ **dann**
      seien $A_j \to w_1, \ldots, A_j \to w_m$ alle Produktionen mit A_j als linker Seite;
      $P := P \cup \{A_i \to w_1, \ldots, A_i \to w_m\}$;
      $P := P \setminus \{A_i \to A_j\}$;
    Ende
  Ende
  Gib die so entstandene Grammatik als G' aus;
Ende

Beachte, dass die Zyklen durch eine Tiefensuche auf D gefunden werden können.

Satz 5.2.4. *Algorithmus 5 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 ε-Produktionen hat, dann hat auch G' keine ε-Produktionen.*

Beweis. Wir zeigen:

1. Das Entfernen eines Zyklus verändert die erzeugte Sprache nicht.
2. Das Entfernen einer Einheitsproduktion $A_i \to A_j$ in der rückwärts-laufenden „Für“-Schleife ändert die erzeugte Sprache nicht.

Wir beweisen beide Teile unabhängig voneinander.

1. Sei $G = (V, \Sigma, P, S)$ die Grammatik mit Zyklus $A_1 \to A_2, \ldots, A_{n-1} \to A_n, A_n \to A_1$ und $G' = (V', \Sigma, P', S')$ die Grammatik nachdem der Zyklus entfernt wurde. Sei σ die

Substitution $\{A_i \mapsto A_1 \mid i \in \{2, \ldots, n\}\}$.

- „$L(G) \subseteq L(G')$“: 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 $A_i \to A_j$ 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 $B \to w$, die nicht vom Zyklus stammt, dann gibt es die Regel $\sigma(B) \to \sigma(w)$ 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 $A_j \to w_j \in P$. Dann gilt $uA_1v \Rightarrow_G \cdots \Rightarrow_G uA_jv \Rightarrow_G uw_jv \Rightarrow_G^* u\sigma(w_j)v \Rightarrow^* w_0$.
  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 Entfernens der Einheitsproduktionen folgt aus Lemma 5.1.1.

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 $A_i \to A_j$ entfernt wird, wurden bereits alle Einheitsproduktionen $A_j \to A_k$ entfernt, d.h. alle rechten Seiten $w_1, \ldots, w_m$ der Produktionen für A_j können nicht nur aus einer Variablen bestehen.

Der zweite Teil des Satzes gilt, da sämtliche Schritte keine ε-Produktionen einführen, sofern die Eingabegrammatik keine ε-Produktionen enthält. □

5.2.2. ⋆ Herstellen der Chomsky-Normalform

Nach Entfernen der ε-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 $A_a \to a$ hinzufügen. Anschließend sind alle Regeln von der Form $A \to a$ oder Regeln der Form $A \to B_1 \cdots B_m$ 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 6 formalisiert diese Schritte.

Algorithmus 6 : 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 ε-Produktionen in G mit Algorithmus 1 und entferne anschließend die Einheitsproduktionen mit Algorithmus 5;
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 die Variable */
$G' := (V' \cup \{A_a\}, \Sigma, \{A \to w[A_a/a] \mid A \to w \in P' \text{ und } |w| > 1\}$
$\cup \{A \to w \mid A \to w \in P' \text{ und } |w| = 1\} \cup \{A_a \to a\}, S)$
Ende
/* Nun sind alle Regeln von der Form $A \to a$ oder $A \to B_1 \cdots B_m$ mit $m \geq 2$ */
für alle $A \to B_1 \cdots B_m \in P'$ *mit* $m > 2$ **tue**
Seien $C_1, \ldots, C_{m-2}$ neue Variablen;
$V' := V' \cup \{C_1, \ldots, C_{m-2}\}$;
/* Ersetze in P' die Produktion $A \to B_1 \cdots B_m$ durch neue Regeln */
$P' := (P' \setminus \{A \to B_1 \cdots B_m\})$
$\cup \{A \to B_1 C_1\} \cup \{C_i \to B_{i+1} C_{i+1} \mid \text{für } i = 1, \ldots, m-3\}$
$\cup \{C_{m-2} \to B_{m-1} B_m\}$;
Ende
Ende

Theorem 5.2.5. *Für CFGs G mit $\varepsilon \notin L(G)$ berechnet Algorithmus 6 eine äquivalente CFG in Chomsky-Normalform.*

Beweis. Die Äquivalenz der erzeugten Sprache ergibt sich aus Satz 3.2.5 für das Entfernen der ε-Produktionen, aus Satz 5.2.4 und aus Lemma 5.1.2 für die weiteren Schritte des Algorithmus (Einführen von Produktionen für $A \to a$ und $A \to B_1 \cdots B_m$). Schließlich verifiziert man leicht, dass die Ausgabe-Grammatik in Chomsky-Normalform ist. □

Beispiel 5.2.6. *Wir verwenden Algorithmus 6 zum Berechnen der Chomsky-Normalform für die Grammatik $G_0 = (\{A, B, C, D, S\}, \{0, 1\}, P_0, S)$ mit*

$$P_0 = \{S \to 1A,\ A \to AB,\ A \to DA,\ A \to \varepsilon,\ B \to 0,\ B \to 1,\ C \to AAA,\ D \to 1AC\}$$

Entfernen der ε-Produktionen (siehe Beispiel 3.2.8) liefert die Grammatik $G_1 = (V, \Sigma, P_1, S)$ mit

$$\begin{aligned} P_1 = \{&S \to 1A,\ S \to 1,\ A \to AB,\ A \to B,\ A \to DA,\ A \to D,\ B \to 0,\ B \to 1,\\ &C \to AAA,\ C \to AA,\ C \to A,\ D \to 1AC,\ D \to 1A,\ D \to 1C,\ D \to 1\} \end{aligned}$$

Wir wenden Algorithmus 5 an, um Einheitsproduktionen zu entfernen. Der gerichtete Graph ist $\mathsf{D} = (\{S, A, B, C, D\}, \{(A, A), (A, B), (A, D), (C, A)\})$ und hat keine Zyklen. Wir erhalten daher:

$$\begin{aligned} P_2 = \{&S \to 1A,\ S \to 1,\ A \to AB,\ A \to B,\ A \to DA,\ A \to D,\ B \to 0,\ B \to 1,\\ &C \to AAA,\ C \to AA,\ C \to A,\ D \to 1AC,\ D \to 1A,\ D \to 1C,\ D \to 1\} \end{aligned}$$

Topologisches Sortieren und Umbenennen der Variablen, sodass „$A_i \to A_j$ impliziert $i < j$“ gilt, erfordert eine Umbenennung, welche die Beziehungen „$A < B$“, „$A < D$“, „$C < A$“ erzeugt. Wir wählen die Umbenennung ρ 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

$$\begin{aligned} P_3 = \{&A_5 \to 1A_2,\ A_5 \to 1,\ A_2 \to A_2A_3,\ A_2 \to A_3,\ A_2 \to A_4A_2,\ A_2 \to A_4,\ A_3 \to 0,\\ &A_3 \to 1,\ A_1 \to A_2A_2A_2,\ A_1 \to A_2A_2,\ A_1 \to A_2,\ A_4 \to 1A_2A_1,\ A_4 \to 1A_2,\\ &A_4 \to 1A_1,\ A_4 \to 1\} \end{aligned}$$

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 $A_i \to A_j$. Für $i = 2$, wird $A_2 \to A_3$ ersetzt durch $A_2 \to 0,\ A_2 \to 1$, und $A_2 \to A_4$ wird ersetzt durch $A_2 \to 1A_2A_1,\ A_2 \to 1A_2,\ A_2 \to 1A_1$ und $A_2 \to 1$. Danach ist

$$\begin{aligned} P_4 = \{&A_5 \to 1A_2,\ A_5 \to 1,\ A_2 \to A_2A_3,\ A_2 \to 0,\ A_2 \to 1,\ A_2 \to A_4A_2,\ A_2 \to 1A_2A_1,\\ &A_2 \to 1A_2,\ A_2 \to 1A_1,\ A_3 \to 0,\ A_3 \to 1,\ A_1 \to A_2A_2A_2,\ A_1 \to A_2A_2,\ A_1 \to A_2,\\ &A_4 \to 1A_2A_1,\ A_4 \to 1A_2,\ A_4 \to 1A_1,\ A_4 \to 1\} \end{aligned}$$

Für $i = 1$ wird $A_1 \to A_2$ ersetzt durch $A_1 \to A_2A_3$, $A_1 \to 0$, $A_1 \to 1$, $A_1 \to A_4A_2$, $A_1 \to 1A_2A_1$, $A_1 \to 1A_2$ und $A_1 \to 1A_1$. 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

$$\begin{aligned} P_5 = \{&A_5 \to 1A_2,\ A_5 \to 1,\ A_2 \to A_2A_3,\ A_2 \to 0,\ A_2 \to 1,\ A_2 \to A_4A_2, A_2 \to 1A_2A_1,\\ &A_2 \to 1A_2,\ A_2 \to 1A_1,\ A_3 \to 0,\ A_3 \to 1,\ A_1 \to A_2A_2A_2,\ A_1 \to A_2A_2,\ A_1 \to A_2A_3,\\ &A_1 \to 0,\ A_1 \to 1,\ A_1 \to A_4A_2,\ A_1 \to 1A_2A_1,\ A_1 \to 1A_2,\ A_1 \to 1A_1,\ A_4 \to 1A_2A_1,\\ &A_4 \to 1A_2,\ A_4 \to 1A_1,\ A_4 \to 1\} \end{aligned}$$

Nun wird Algorithmus 6 fortgesetzt und zunächst werden alle Terminalsymbole durch neue Produktionen dargestellt: Dies seien $B_0 \to 0$ und $B_1 \to 1$. Ersetzen aller Vorkommen von 0 durch B_0 und 1 durch B_0 in rechten Seiten mit Wortlänge > 1 ergibt daher:

$$\begin{aligned} P_6 = \{&B_0 \to 0,\ B_1 \to 1,\ A_5 \to B_1A_2,\ A_5 \to 1,\ A_2 \to A_2A_3,\ A_2 \to 0,\ A_2 \to 1,\ A_2 \to A_4A_2,\\ &A_2 \to B_1A_2A_1,\ A_2 \to B_1A_2,\ A_2 \to B_1A_1,\ A_3 \to 0,\ A_3 \to 1,\ A_1 \to A_2A_2A_2,\\ &A_1 \to A_2A_2,\ A_1 \to A_2A_3,\ A_1 \to 0,\ A_1 \to 1,\ A_1 \to A_4A_2,\ A_1 \to B_1A_2A_1,\\ &A_1 \to B_1A_2,\ A_1 \to B_1A_1,\ A_4 \to B_1A_2A_1,\ A_4 \to B_1A_2,\ A_4 \to B_1A_1,\ A_4 \to 1\} \end{aligned}$$

Schließlich werden alle rechten Seiten mit Länge ≥ 3 durch Einführen neuer Variablen zerlegt, sodass wir als Ausgabegrammatik $G_7 = (V_7, \Sigma, P_7, A_5)$ in Chomsky-Normalform erhalten, wobei

$$V_7 = \{A_1, A_2, A_3, A_4, A_5, B_0, B_1, C_1, C_2, C_3, C_4\}$$

$$\begin{aligned} P_7 = \{&B_0 \to 0,\ B_1 \to 1,\ A_5 \to B_1A_2,\ A_5 \to 1,\ A_2 \to A_2A_3,\ A_2 \to 0,\ A_2 \to 1,\ A_2 \to A_4A_2,\\ &A_2 \to B_1C_1\ C_1 \to A_2A_1,\ A_2 \to B_1A_2,\ A_2 \to B_1A_1,\ A_3 \to 0,\ A_3 \to 1,\\ &A_1 \to A_2C_2,\ C_2 \to A_2A_2,\ A_1 \to A_2A_2,\ A_1 \to A_2A_3,\ A_1 \to 0,\ A_1 \to 1,\ A_1 \to A_4A_2,\\ &A_1 \to B_1C_3,\ C_3 \to A_2A_1,\ A_1 \to B_1A_2,\ A_1 \to B_1A_1,\ A_4 \to B_1C_4,\ C_4 \to A_2A_1,\\ &A_4 \to B_1A_2,\ A_4 \to B_1A_1,\ A_4 \to 1\} \end{aligned}$$

5.3. ⋆ Greibach-Normalform

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

Definition 5.3.1. *Ein CFG $G = (V, \Sigma, P, S)$ mit $\varepsilon \notin L(G)$ ist in* Greibach-Normalform, *falls alle Produktionen in P von der Form $A \to aB_1B_2 \dots B_j$ mit $j \geq 0$, $A, B_1, \dots, B_j \in V$ und $a \in \Sigma$ sind.*

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 ε-Produktionen) ist in Greibach-Normalform.

5.3.1. Herstellen der Greibach-Normalform

Algorithmus 7 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 $A_i \to A_j u$ 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 5.1.1. Für den Fall $A_i \to A_i u$ wird die Elimination der Links-Rekursion, wie wir sie in Lemma 5.1.3 gesehen haben, angewendet. Nach Abschluss der geschachtelten „Für"-Schleifen, sind Regeln der Form $A_i \to A_j u$ nur noch möglich, wenn $j > i$ gilt. Daher muss jede Regel mit A_n (der „größten Variablen") als linker Seite von der Form $A_n \to u$ sein, wobei u nicht mit einer Variablen (und daher mit einem Symbol aus Σ) beginnt. Deshalb können wir sukzessive die Regeln für A_n, dann für A_{n-1}, u.s.w. hineinkopieren in Regel der Form $A_i \to A_j u$ (und A_j durch alle Möglichkeiten ersetzen): Nach dem Behandeln von A_n, kommt kein A_n mehr in der Form $A_i \to A_n u$ vor, nach dem Behandeln von A_{n-1} kommt kein A_{n-1} mehr in der Form $A_i \to A_{n-1} u$ vor u.s.w., sodass am Ende keine Regeln mehr die Form $A_i \to A_j u$ hat. Die Korrektheit der Ersetzungen folgt aus Lemma 5.1.1.

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 5.1.1.

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 $A \to au$ 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 7 in Greibach-Normalform transformieren. D.h. wir haben den folgenden Satz gezeigt:

Satz 5.3.2. *Zu jeder CFG G mit $\varepsilon \notin L(G)$ gibt es eine CFG G' in Greibach-Normalform, sodass $L(G') = L(G)$ gilt.*

Algorithmus 7 : Herstellen der Greibach-Normalform

Eingabe : CFG $G = (\{A_1, \ldots, A_n\}, \Sigma, P, A_i)$ 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 $A_i \to A_j u$ nur gibt, wenn $j > i$ gilt. */
für $i = 1$ *bis* n **tue**
für $j = 1$ *bis* $i - 1$ **tue**
/* Ersetzen der Regeln $A_i \to A_j u$ mit $i > j$ */
für alle $A_i \to A_j u \in P$ **tue**
Seien $A_j \to w_1 \mid \cdots \mid w_m$ alle Regeln in P mit A_j als linker Seite;
Ersetze $A_i \to A_j u$ durch $A_i \to w_1 u \mid \ldots \mid w_m u$ in P;
Ende
Ende
/* Ersetzen der Regeln $A_i \to A_i u$ */
wenn $A_i \to A_i u \in P$ **dann**
Wende Lemma 5.1.3 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 $A_i \to A_j u$ mit $j > i$. */
/* Insbesondere gilt damit auch für alle Regeln $A_n \to u$: u beginnt mit einem Zeichen aus Σ */
/* Ziel der nächsten Schleife: Ersetze alle Regeln $A_i \to A_j u$ mit $j > i$ */
für $i = n - 1$ *bis* 1 **tue**
für alle $A_i \to A_j u \in P$, $j > i$ **tue**
Seien $A_j \to w_1 \mid \cdots \mid w_m$ alle Regeln mit A_j als linker Seite;
/* Beachte: Alle $w_1, \ldots, w_m$ fangen mit Zeichen aus Σ an, da Schleife absteigend läuft */
Ersetze $A_i \to A_j u$ durch $A_i \to w_1 u \mid \cdots \mid w_m u$ in P;
Ende
Ende
/* Nun sind alle Regeln mit A_i, $i = 1, \ldots, n$ 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 $B_i \to A_j u \in P$ **tue**
Seien $A_j \to w_1 \mid \cdots \mid w_m$ alle Regeln mit A_j als linker Seite;
Ersetze $B_i \to A_j u$ durch $B_i \to w_1 u \mid \cdots \mid w_m u$ in P;
Ende
Ende
Ende

Beispiel 5.3.3. *Betrachte* $G = (\{A_1, A_2, A_3\}, \{a, b, c, d\}, P, A_1)$ *mit*

$$\begin{aligned} P = \{&A_1 \to A_1A_2 \mid a,\\ &A_2 \to A_1A_3 \mid A_3A_3,\\ &A_3 \to A_3c \mid d\} \end{aligned}$$

Beim Durchlaufen der doppelt-geschachtelten „Für"-Schleifen, wird zunächst für $i = 1$ *und* $j = 1$ *die Produktion* $A_1 \to A_1A_2$ *ersetzt durch* $A_1 \to aB_1 \mid a$ *und* $B_1 \to A_2 \mid A_2B_1$*, d.h. danach ist die Menge der Produktionen*

$$\begin{aligned} P_1 = \{&A_1 \to aB_1 \mid a,\\ &A_2 \to A_1A_3 \mid A_3A_3,\\ &A_3 \to A_3c \mid d,\\ &B_1 \to A_2 \mid A_2B_1\} \end{aligned}$$

Für $i = 2$ *und* $j = 1$ *wird die Produktion* $A_2 \to A_1A_3$ *ersetzt durch* $A_2 \to aB_1A_3 \mid aA_3$*, d.h. die Menge der Produktionen ist danach*

$$\begin{aligned} P_2 = \{&A_1 \to aB_1 \mid a,\\ &A_2 \to aB_1A_3 \mid aA_3 \mid A_3A_3,\\ &A_3 \to A_3c \mid d,\\ &B_1 \to A_2 \mid A_2B_1\} \end{aligned}$$

Für $i = 3$ *und* $j = 3$ *wird* $A_3 \to A_3c$ *ersetzt durch* $A_3 \to d \mid dB_3$, $B_3 \to c \mid cB_3$*, d.h. die Menge der Produktionen ist danach*

$$\begin{aligned} P_3 = \{&A_1 \to aB_1 \mid a,\\ &A_2 \to aB_1A_3 \mid aA_3 \mid A_3A_3,\\ &A_3 \to d \mid dB_3,\\ &B_1 \to A_2 \mid A_2B_1,\\ &B_3 \to c \mid cB_3\} \end{aligned}$$

Durchlaufen der „Für $i = n - 1$ *bis* 1*"-Schleife ersetzt* $A_2 \to A_3A_3$ *durch* $A_2 \to dA_3 \mid dB_3A_3$*, d.h. die Menge der Produktionen ist danach*

$$\begin{aligned} P_4 = \{&A_1 \to aB_1 \mid a,\\ &A_2 \to aB_1A_3 \mid aA_3 \mid dA_3 \mid dB_3A_3,\\ &A_3 \to d \mid dB_3,\\ &B_1 \to A_2 \mid A_2B_1,\\ &B_3 \to c \mid cB_3\} \end{aligned}$$

Durchlauf der letzten „Für"-Schleife ersetzt

- $B_1 \to A_2$ *durch* $B_1 \to aB_1A_3 \mid aA_3 \mid dA_3 \mid dB_3A_3$*, und*
- $B_1 \to A_2B_1$ *durch* $B_1 \to aB_1A_3B_1 \mid aA_3B_1 \mid dA_3B_1 \mid dB_3A_3B_1$

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*

$$\begin{aligned} P' = \{&A_1 \to aB_1 \mid a,\\ &A_2 \to aB_1A_3 \mid aA_3 \mid dA_3 \mid dB_3A_3,\\ &A_3 \to d \mid dB_3,\\ &B_1 \to aB_1A_3 \mid aA_3 \mid dA_3 \mid dB_3A_3 \mid aB_1A_3B_1 \mid aA_3B_1 \mid dA_3B_1 \mid dB_3A_3B_1,\\ &B_3 \to c \mid cB_3\} \end{aligned}$$

als Ausgabe (in Greibach-Normalform) erzeugt.

5.4. ⋆ Widerlegen der Kontextfreiheit

In diesem Abschnitt lernen wir Hilfsmittel kennen, um zu zeigen, dass formale Sprachen nicht kontextfrei sind.

Als Vorüberlegung beachte, dass ein Binärbaum (ein Baum für den jeder Knoten 2 oder 0 Kinder hat) mit $> 2^k$ Blättern stets einen Pfad der Länge $> k$ 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 einen Pfad der Länge $\geq k - 1$ hat. Verwende diesen Pfad und verlängere ihn um die Wurzel des gesamten Baums. Der Pfad hat Länge $\geq k$.

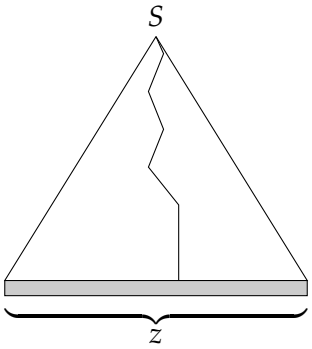
Andersrum gilt, dass ein Binärbaum, von dem alle Pfade eine Länge $\leq k$ haben, maximal $\leq 2^k$ Blätter hat. Dies beweisen wir mit Induktion über k: Für $k = 0$ besteht der Baum nur aus $2^0 = 1$ Blatt. Wenn $k > 0$, dann hat der Baum zwei Teilbäume unter der Wurzel, von denen alle Pfade eine Länge $\leq k - 1$ haben. Durch die Induktionshypothese haben die beiden Teilbäume jeweils $\leq 2^{k-1}$ Blätter. Der Baum als Ganzes hat dann $\leq 2^{k-1} + 2^{k-1} = 2^k$ Blätter.

Wir werden diese beiden Eigenschaften im Beweis des folgenden Pumping-Lemmas für kontextfreie Sprache verwenden. Ähnlich zum Pumping-Lemma für reguläre Sprachen (siehe Lemma 4.9.1), liefert dass Pumping-Lemma für CFLs ein 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 5.4.1 (Pumping-Lemma für CFLs)**.** *Sei L eine kontextfreie Sprache. Dann gibt es eine Zahl $n \in \mathbb{N}_{>0}$, sodass jedes Wort $z \in L$, das Mindestlänge n hat (d.h. $|z| \geq n$), als $z = uvwxy$ geschrieben werden kann, sodass gilt:*

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

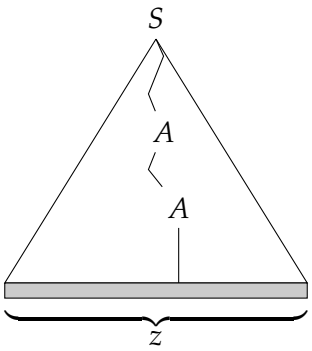
Beweis. Sei L eine kontextfreie Sprache und sei $G = (V, \Sigma, P, S)$ eine CFG für $L \setminus \{\varepsilon\}$ in Chomsky-Normalform. Betrachte eine Ableitung und den zugehörigen Syntaxbaum eines Wortes z mit $|z| \geq 2^{|V|} = n$:



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 der Form $A \to a$ anwendet).

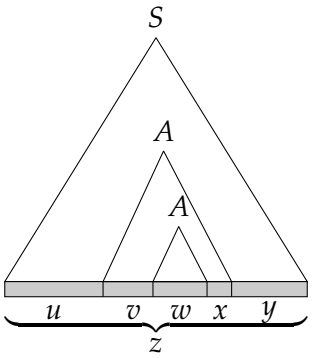
Der Baum, ohne diesen letzten Schritt ist ein Binärbaum mit $|w| \geq 2^{|V|}$ Blättern, der daher einen Pfad der Länge $\geq |V|$ haben muss, der $|V| + 1$ Variablen benutzt. Unter den Pfaden der Länge $\geq |V|$ wählen wir einen maximaler Länge aus.

Das 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.

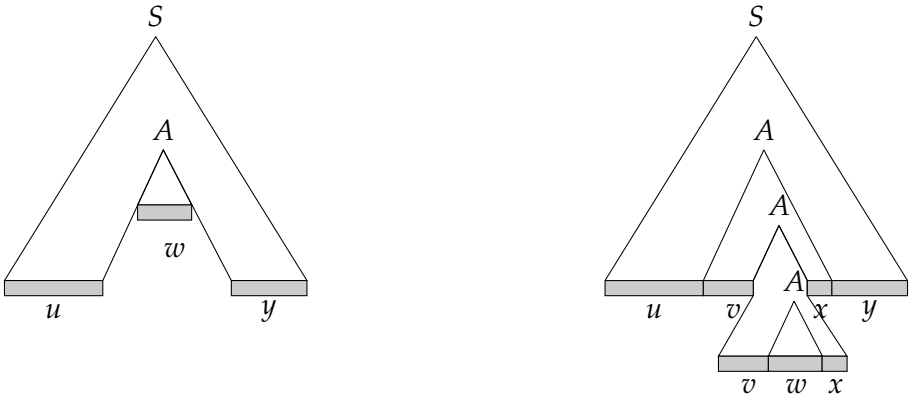
Wenn wir nun die beiden Teilbäume betrachten, die jeweils A als Wurzel haben, dann entsprechen diese den Ableitungen von Teilworten von z, wobei das tiefere A ein Teilwort des Wortes des vom oberen A erzeugten Wortes ist. D.h. wir können das Wort z zerlegen in $z = uvwxy$, wobei w das vom unteren A erzeugte Teilwort ist, vwx 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 und daraus folgt, dass $|v| \geq 1$ oder $|x| \geq 1$ (oder beides), vereinfacht geschrieben $|vx| \geq 1$, gelten muss.

Da wir die tiefsten Vorkommen der gleichen Variablen A gewählt haben, muss der Pfad vom oberen A bis zur Blattebene eine Länge $\leq |V|$ haben. Da der Pfad von der Wurzel maximaler Länge ist, müssen andere Pfade vom oberen A bis zur Blattebene kürzer oder gleich lang sein. Daraus folgt 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 $S \Rightarrow^* uv^iwx^iy$ für alle $i \in \mathbb{N}$. Im Baum dargestellt für die Fälle $i = 0$ und $i = 2$:



□

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.

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 $n \in \mathbb{N}_{>0}$.
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 $uv^iwx^iy \notin L$.

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

Satz 5.4.2. *Die Sprache $L = \{a^lb^lc^l \mid l \in \mathbb{N}\}$ ist nicht kontextfrei.*

Beweis. Wir verwenden das Pumping-Lemma für CFGs. Sei $n \in \mathbb{N}_{>0}$. Wir wählen $z = a^nb^nc^n$. Für jede Zerlegung $z = uvwxy$ mit $|vx| \geq 1$ und $|vwx| \leq n$ ist vwx von der Form a^ib^j oder b^ic^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 $uwy \notin L$. □

Satz 5.4.3. *Die Sprache $L = \{a^ib^jc^id^j \mid i, j \in \mathbb{N}_{>0}\}$ ist nicht kontextfrei.*

Beweis. Wir verwenden das Pumping-Lemma für CFGs. Sei $n \in \mathbb{N}_{>0}$. Wir wählen $z = a^nb^nc^nd^n$. 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 $i, j \in \mathbb{N}$ 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 $j' < n$. □

Folgener Satz wird in der Vorlesung nicht behandelt und ist daher kein Prüfungsstoff.

Satz 5.4.4. *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 Σ. Das Pumping-Lemma für CFLs zeigt: Es gibt eine Zahl $n \in \mathbb{N}_{>0}$, 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 $i \in \mathbb{N}$: $uv^i wx^i y \in L$.

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 $i \in \mathbb{N}$: $uwyv^i x^i \in L$.

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 $i \in \mathbb{N}$: $a^k a^{i \cdot l} \in L$.

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

Sei $\widetilde{L}_j := \{a^j a^{i \cdot n!} \mid i \in \mathbb{N}\}$. Die Sprache $\widetilde{L}_j$ (für festes n und j) ist regulär, denn der NFA mit ε-Übergängen $M_j = (\{z_0, \ldots, z_{n!+j}\}, \Sigma, \delta, z_0, \{z_{n!+j}\})$ 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 $\widetilde{L}_j$.

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

> Sei $j \geq n!$ und $a^j \in L$. Dann ist für alle $i \in \mathbb{N}$ 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 $i \in \mathbb{N}$: $a^j a^{i \cdot n!} = a^{k+l+i \cdot l \cdot (n!/l)} = a^k a^{l \cdot (1 + i + (n!/l))}$ und daher ist $a^j a^{i \cdot n!} \in L$.

Umgekehrt gilt für jedes Wort $a^m \in L$ mit $m \geq n!$: $a^m \in \widetilde{L}_j$ (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 $\widetilde{L}_j$ liegen, und zudem, dass wir nur endlich viele solcher Mengen $\widetilde{L}_j$ 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 $q \in \mathbb{N}$, sodass die unendliche Vereinigung $\bigcup\{\widetilde{L}_j \mid a^j \in L, j \in \mathbb{N}, j \geq n!\}$ als *endliche* Vereinigung

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

dargestellt werden kann.

Nun können wir die gesamte Sprache L darstellen durch:

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

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. □

Satz 5.4.5. *Die Sprachen*

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

sind allesamt nicht kontextfrei.

Beweis. Für alle vier Sprachen haben gezeigt, dass sie nicht regulär sind (Sätze 4.9.4 bis 4.9.6 und 4.12.5), alle sind über dem unären Alphabet $\Sigma = \{a\}$ definiert. Daher folgt mit Satz 5.4.4, dass die Sprachen auch nicht kontextfrei sind. □

Der Rest dieses Unterabschnitts (Ogdens Lemma und dessen Konsequenzen und Anwendungen) wird in der Vorlesung nicht behandelt und ist daher kein Prüfungsstoff.

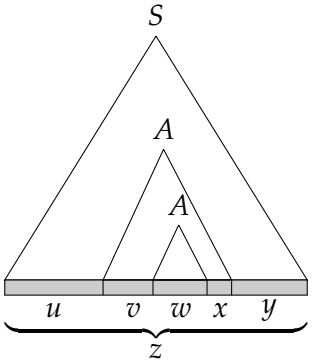
Eine Verallgemeinerung des Pumping-Lemmas für CFLs ist das sogenannte Ogdens Lemma (Ogd68) (benannt nach William Ogden)

Lemma 5.4.6 (Ogdens Lemma)**.** *Sei L eine kontextfreie Sprache. Dann gibt es eine Zahl $n \in \mathbb{N}_{>0}$, sodass jedes Wort $z \in L$, das Mindestlänge n hat und in dem mindestens n Zeichen markiert sind, als $z = uvwxy$ geschrieben werden kann, sodass gilt:*

- *vx enthält mindestens ein markiertes Zeichen*
- *vwx enthält höchstens n markierte Zeichen*
- *für alle $i \geq 0$: $uv^iwx^iy \in L$.*

Beweis. Sei L kontextfrei und sei $G = (V, \Sigma, P, S)$ eine CFG in Chomsky-Normalform mit $L(G) = L \setminus \{\varepsilon\}$. Betrachte eine Ableitung und den zugehörigen Syntaxbaum eines Wortes z, sodass $|z| \geq 2^{|V|+1} = n$ ist. Wir nehmen an, dass $\geq n$ Zeichen in z markiert sind. 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 der Form $A \to a$ anwendet). Wir wählen einen Pfad im Syntaxbaum von der Wurzel bis zu einem Blatt nach folgendem Verfahren aus: An einem Knoten A mit Kindern B und C (d.h. eine Produktion $A \to BC$ wurde angewendet), der die Wurzel des Teilbaums zur Erzeugung von $w_A = w_Bw_C$ ist, wähle B wenn w_B mehr markierte Zeichen enthält als w_C, anderenfalls wähle C. Wir nennen A einen Verzweigungsknoten,

wenn w_B und w_C jeweils noch markierte Zeichen enthalten. Nach jedem Verzweigungsknoten ist die Menge der noch markierten Zeichen im Wort allerhöchstens halbiert (da wir den Pfad so wählen, dass wir das Teilwort mit meisten Markierungen wählen). Daher gibt es mindestens $|V| + 1$ Verzweigungsknoten auf dem Pfad. Wir betrachten den untersten Teilpfad davon (in einem Blatt endend), der genau $|V| + 1$ Verzweigungsknoten hat. Auch auf diesem Pfad wird eine Variable $A \in V$ doppelt besucht. Die mit A markierten Knoten liegen übereinander und induzieren daher das Bild



und eine Zerlegung $z = uvwxy$, sodass das oberere A den Teilbaum mit Blattmarkierungen vwx und das untere A den Teilbaum mit Blattmarkierungen w erzeugt. Sei T_{vwx} der Teilbaum mit dem oberen A als Wurzel und T_w der Teilbaum mit dem unteren A als Wurzel. Der untere Teilbaum T_w kann nur im rechten oder im linken Unterbaum vom T_{vwx} liegen.

Aber beide Unterbäume enthalten noch markierte Zeichen (da das obere A ein Verzweigungsknoten ist). Daher folgt, dass vx mindestens ein markiertes Zeichen enthält.

Der Pfad, den wir betrachten hat $|V| + 1$ Verzweigungsknoten. Daher hat der Pfad vom oberen A aus ebenfalls maximal $|V| + 1$ Verzweigungsknoten. Damit können wir maximal $2^{|V|+1} = n$ Zeichen markieren und daher hat das vom oberen A erzeugte Wort vwx höchstens n markierte Zeichen.

Schließlich gilt $uv^iwx^iy \in L(G)$ durch die gegebenen Ableitungen $S \Rightarrow_G^* uAv$, $A \Rightarrow_G^* vAx$ und $A \Rightarrow_G^* w$. □

Satz 5.4.7. *Das Pumping-Lemma für CFLs ist ein Spezialfall von Ogdens Lemma.*

Beweis. Markiere alle Zeichen in z (anstelle von mindestens n), dann erhält man das Pumping-Lemma für CFLs. □

Beispiel 5.4.8. *Die Sprache $L = \{a^ib^jc^jd^j \mid i \in \mathbb{N}_{>0}, j \in \mathbb{N}\} \cup (\{b\}^*\{c\}^*\{d\}^*)$ erfüllt die Pumping-Eigenschaft für CFLs. Wähle $n = 4$, sei $z \in L$ mit $|z| \geq 4$ mit $z = a_1 \cdots a_{|z|}$. Wenn $z \in (\{b\}^*\{c\}^*\{d\}^*)$ dann wähle die Zerlegung $u = v = w = \varepsilon$, $x = a_1$ $y = a_2 \cdots a_{|z|}$. Dann gilt $z = uvwxy$ und $uv^iwx^iy = a_1^iy \in (\{b\}^*\{c\}^*\{d\}^*)$ für alle $i \in \mathbb{N}$. Wenn $z \in \{a^ib^jc^jd^j \mid i \in \mathbb{N}_{>0}, j \in \mathbb{N}\}$ dann wähle die Zerlegung $u = v = w = \varepsilon$, $x = a_1 = a$, $y = a_2 \cdots a_{|z|}$. Dann gilt $z = uvwxy$ und $uv^iwx^iy = a^iy \in L$ für alle $i \in \mathbb{N}$.*

Beispiel 5.4.8 liefert eine Beispielsprache, für welche die Pumping-Eigenschaft erfüllt ist, die aber nicht kontextfrei ist, was sich mit Ogdens Lemma zeigen lässt:

Satz 5.4.9. *Die Sprache $L = \{a^ib^jc^jd^j \mid i \in \mathbb{N}_{>0}, j \in \mathbb{N}\} \cup (\{b\}^*\{c\}^*\{d\}^*)$ ist nicht kontextfrei.*

Beweis. Wir verwenden Ogdens Lemma (Lemma 5.4.6). Sei $n \in \mathbb{N}_{>0}$. Wir wählen das Wort $z = ab^nc^nd^n \in L$ und markieren das Teilwort bc^nd. Sei $z = uvwxy$, wobei vx mindestens eines und vwx höchstens n markierte Zeichen enthält. Dann enthält das Teilwort vwx nicht alle Zeichen b,c und d (da es ansonsten $n+2$ markiert Zeichen enthielte). Daher ist $uv^2wx^2y \notin L$, da dort die Anzahl an b's, c's, und d's nicht gleich sein kann, aber uv^2wx^2y mit einem a beginnt. □

5.5. ⋆ Abschlusseigenschaften kontextfreier Sprachen

Theorem 5.5.1. *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 $G_\cup = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}, S)$. 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 $G_\cdot$ die CFG definiert durch $G_\cdot = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1S_2\}, S)$. Offensichtlich gilt $L(G_\cdot) = L(G_1)L(G_2) = L_1L_2$. 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 $S_0 \to S$). Sei $S' \notin V$ und sei $G_* = (V \cup \{S'\}, \Sigma, (P \setminus \{S \to \varepsilon\}) \cup \{S' \to \varepsilon, S' \to S, S \to SS\}, S')$. Dann gilt $L(G_*) = L(G)^*$. Das zeigt, dass kontextfreie Sprachen bezüglich dem Kleeneschen Stern-Operator (der reflexiv-transitiven Hülle) abgeschlossen sind.

Die Sprachen $L_1 = \{a^nb^mc^m \mid m, n \in \mathbb{N}\}$ und $L_2 = \{a^mb^mc^n \mid m, n \in \mathbb{N}\}$ sind beide kontextfrei, da sie von den CFGs $G_1 = (\{A, D, S\}, \{a, b, c\}, \{S \to AD, A \to \varepsilon \mid aA, D \to bDc \mid \varepsilon\}, S)$ und $G_2 = (\{C, D, S\}, \{a, b, c\}, \{S \to DC, C \to cC \mid \varepsilon, D \to aDb \mid \varepsilon\}, S)$ erzeugt werden. Allerdings ist $L_1 \cap L_2 = \{a^nb^nc^n \mid n \in \mathbb{N}_{>0}\}$ nicht kontextfrei, was in Satz 5.4.2 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.

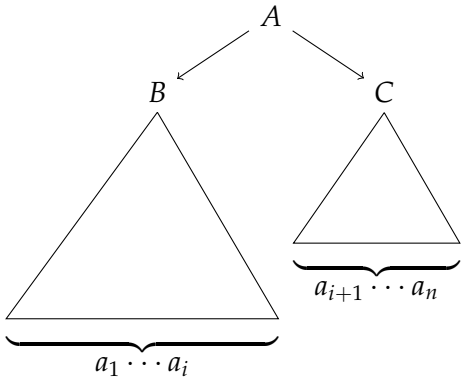
□

5.6. Effiziente Lösung des Wortproblems: Der CYK-Algorithmus

Wir haben bereits einen Algorithmus behandelt (Algorithmus 2), der das Wortproblem für Typ 1-, 2- und 3-Grammatiken 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 Grammatiken 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 $w = a_1 \cdots a_n$ ($n > 0$) erzeugt:
>
> - Wenn $w = a \in \Sigma$, dann prüfe ob $A \to a \in P$.
> - Anderenfalls ($|w| > 1$) kann A nur dann das Wort w erzeugen, wenn es eine Produktion $A \to BC \in P$ 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 $A \to BC \in P$ 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 $A \to a_i$ 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 $A \to BC$, 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 8 gibt den CYK-Algorithmus in Pseudocode 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.

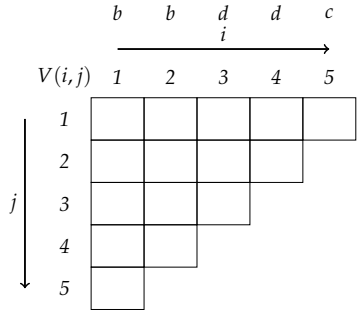
Theorem 5.6.1. *Das Wortproblem für kontextfreie Grammatiken kann in Polynomialzeit entschieden werden.*

Algorithmus 8 : CYK-Algorithmus

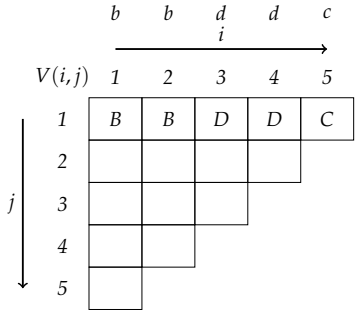
Eingabe : CFG $G = (V, \Sigma, P, S)$ in Chomsky-Normalform und Wort $w = a_1 \cdots a_n \in \Sigma^*$
Ausgabe : `Ja`, wenn $w \in L(G)$ und `Nein`, wenn $w \notin L(G)$
Beginn
  für $i = 1$ *bis* n **tue**
    $V(i,1) = \{A \in V \mid A \to a_i \in P\}$
  Ende
  für $j = 2$ *bis* $n-1$ **tue**
    für $i = 1$ *bis* $n+1-j$ **tue**
      $V(i,j) = \emptyset$;
      für $k = 1$ *bis* $j-1$ **tue**
        $V(i,j) = V(i,j) \cup \{A \in V \mid A \to BC \in P, B \in V(i,k), C \in V(i+k, j-k)\}$
      Ende
    Ende
  Ende
  wenn $S \in V(1,n)$ **dann**
    return `Ja`
  sonst
    return `Nein`
  Ende
Ende

Beispiel 5.6.2. *Sei* $G = (\{S, A, B\}, \{b, c, d\}, P, S)$ *mit* $P = \{S \to AC,\ A \to BE,\ A \to BD,\ E \to AD,\ C \to c,\ B \to b,\ D \to d\}$ *und* $w = bbddc$. *Wir führen den CYK-Algorithmus aus, um zu prüfen, ob* $w \in L(G)$ *gilt:*

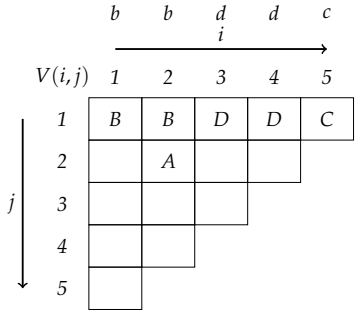
Am Anfang ist die Tabelle $V(i,j)$ *leer:*



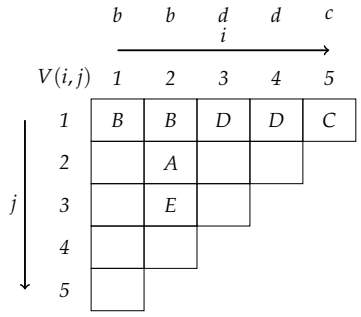
Nach Abarbeiten der ersten Schleife sind alle $V(i,1)$*-Einträge gefüllt:*



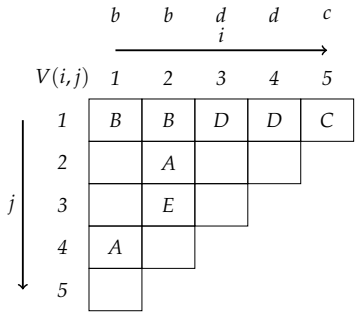
Für $j = 2$ *wird* $A \in V(2,2)$ *eingefügt, da* $A \to BD \in P$, $B \in V(2,1)$ *und* $D \in V(3,1)$*:*



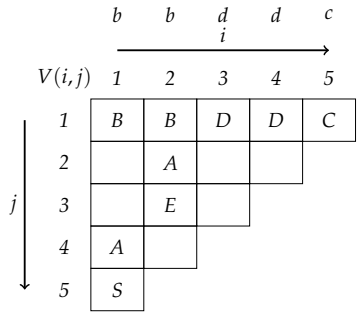
Für $j = 3$ wird $E \in V(2,3)$ eingefügt, da $E \to AD \in P$, $A \in V(2,2)$ und $D \in V(4,1)$:



Für $j = 4$ wird $A \in V(1,4)$ eingefügt, da $A \to BE \in P$, $B \in V(1,1)$ und $E \in V(2,3)$:



Für $j = 5$ wird $S \in V(1,5)$ eingefügt, da $S \to AC \in P$, $A \in V(1,4)$ und $C \in V(5,1)$:



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 5.6.2 ist die zugehörige Linksableitung $S \Rightarrow AC \Rightarrow BEC \Rightarrow bEC \Rightarrow bADC \Rightarrow bBDDC \Rightarrow bbDDC \Rightarrow bbdDC \Rightarrow bbddC \Rightarrow bbddc$.

5.7. 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 $\{w\$\overline{w} \mid w \in \{a,b,c\}^*\}$ 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 $\overline{w}$ 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 und entnommen werden können. Abb. 5.1 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 5.7.1 (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,
- *Σ ist das (endliche)* Eingabealphabet,
- *Γ ist das (endliche)* Kelleralphabet,
- *$\delta : (Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \to \mathcal{P}_e(Z \times \Gamma^*)$ ist die* Zustandsüberführungsfunktion *(oder nur* Überführungsfunktion*)*
- *$z_0 \in Z$ ist der* Startzustand *und*

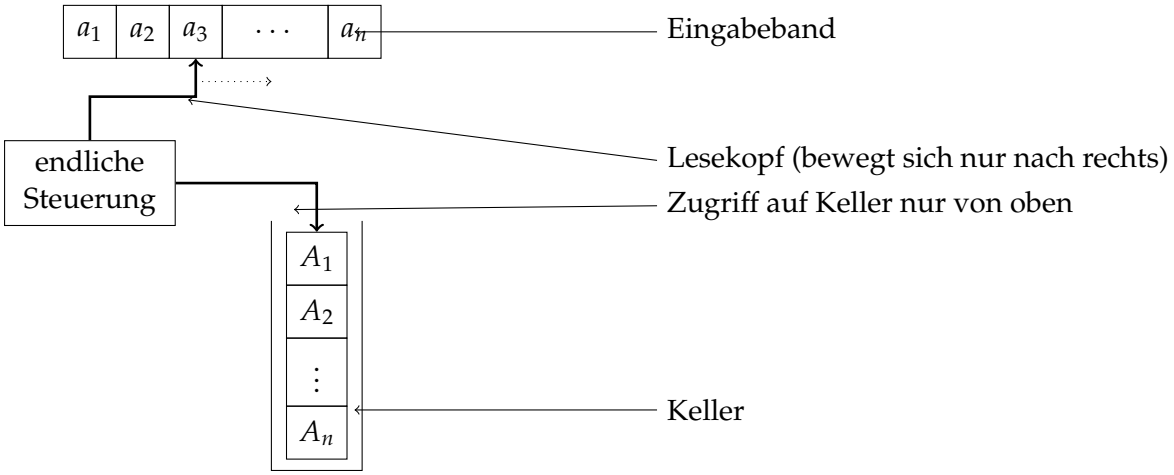


Abbildung 5.1.: Illustration des Kellerautomaten

- $\# \in \Gamma$ *ist das* Startsymbol im Keller.

Wenn $(z', B_1 \cdots B_k) \in \delta(z, a, A)$ 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' übergehen kann und dabei A durch die Zeichenfolge $B_1 \cdots B_k$ ersetzt, sodass B_1 ganz oben auf dem Keller liegt. Abb. 5.2 illustriert diesen Übergang. Beachte, dass $k = 0$ möglich ist und in diesem Fall das Symbol A vom Kellerspeicher gelöscht wird.

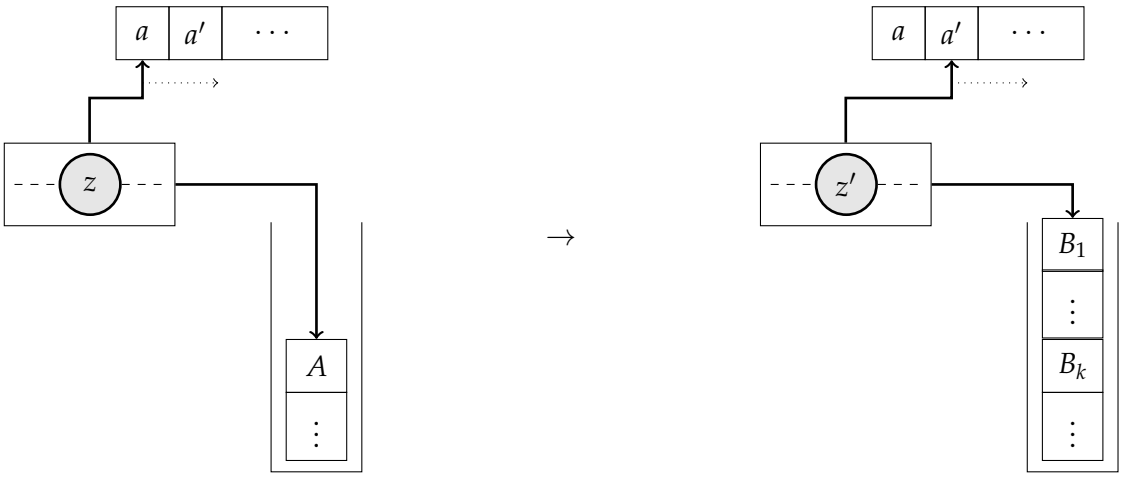


Abbildung 5.2.: Illustration der Ausführung des Schrittes $(z', B_1 \cdots B_k) \in \delta(z, a, A)$

Die so definierten PDAs sind nichtdeterministisch und erlauben ε-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 Buchführen über den aktuellen Zustand, die Resteingabe und den aktuellen Kellerinhalt. Formal wird dies durch eine *Konfiguration* dargestellt:

Definition 5.7.2 (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 5.7.3 (Übergangsrelation 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 5.7.4 (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 5.7.5. *Der folgende PDA $M = (\{z_0, z_1\}, \{a, b\}, \{B, \#\}, \delta, z_0, \#)$ akzeptiert die Sprache $\{a^i b^i \mid i \in \mathbb{N}\}$, wobei wir für δ 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 5.7.6. *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 $L = \{w \in \{a,b\}^* \mid w \text{ ist ein Palindrom}\}$. Da w ein Palindrom sein muss, gibt es im Wesentlichen vier Möglichkeiten $w = \varepsilon$, $w = u\overline{u}$, $w = ua\overline{u}$, oder $w = ub\overline{u}$ (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 $\overline{u}$ 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 $u\overline{u}$ und $uc\overline{u}$ 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 Übergangsfolge, bei der der richtige Zeitpunkt zum Übergang von z_0 zu z_1 verpasst wird:

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

Ein erfolgreicher Lauf mit Eingabe abbba ist:

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

5.7.1. 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 5.7.7 (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,
- *Σ ist das (endliche)* Eingabealphabet,
- *Γ ist das (endliche)* Kelleralphabet,
- *$\delta : (Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \to \mathcal{P}_e(Z \times \Gamma^*)$ ist die* Zustandsüberführungsfunktion *(oder nur* Überführungsfunktion*),*
- *$z_0 \in Z$ 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

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

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

Lemma 5.7.8. *Für jeden Kellerautomat mit Endzuständen M kann ein Kellerautomat M' (ohne Endzustände) konstruiert werden, sodass 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 δ' 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 5.7.9. *Für jeden Kellerautomat M kann ein Kellerautomat mit Endzuständen M' konstruiert werden, sodass $L(M) = L(M')$ 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 $M' = (Z \cup \{z_0', z_E\}, \Sigma, \Gamma \cup \{\$\}, \delta', z_0', \$, \{z_E\})$, sodass δ' 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 5.7.8 und 5.7.9 zeigen:

Satz 5.7.10. *PDAs mit Endzuständen und PDAs ohne Endzustände (mit Akzeptanz durch leeren Keller) sind äquivalente Formalismen.*

5.7.2. 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.

5.7.2.1. ⋆ Kontextfreie Sprachen werden durch Kellerautomaten erkannt

Satz 5.7.11. *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

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

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 $\varepsilon \in L$ g.d.w. $(z_0, \varepsilon, S) \vdash (z_0, \varepsilon, \varepsilon)$ und damit $\varepsilon \in L(M)$.

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

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

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 $B_x \to a_i B_1 \cdots B_j \in P$ als letzte Produktion angewendet wurde. Die Induktionsannahme zeigt, dass $S \Rightarrow_G^{i-1} a_1 \cdots a_{i-1} B_x B_{j+1} \cdots B_m$ g.d.w. $(z_0, a_1 \cdots a_{i-1} w, S) \vdash^{i-1} (z_0, w, B_x B_{j+1} \cdots B_k)$. 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 $B_x \to a_i B_1 \cdots B_j$ 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 $S \Rightarrow_G^{i} a_1 \cdots a_i B_1 \cdots B_k$. □

5.7.2.2. Kellerautomaten, die maximal 2 Symbole auf den Keller legen

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 5.7.12. *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 δ' und Γ' 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 $C_3, \ldots, C_k \in \Gamma'$ 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. □

Bemerkung 5.7.13. *Das letzte Lemma kann auch bewiesen werden, indem statt zusätzlicher Kellersymbole zusätzliche Zustände verwendet werden.*

5.7.2.3. ⋆ Kellerautomaten akzeptieren kontextfreie Sprachen

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 5.7.14. *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 5.7.12 keine Einschränkung ist).

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

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

Wir beweisen im Folgenden die Aussage

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

Da $S \to \langle z_0, \#, z\rangle \in P$ 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 $\langle z', A, z\rangle \to a$ 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 $j + l \leq i - 1$ 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 $\langle z', A, z\rangle \to a \in P$ 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 α 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 $\langle z', A, z\rangle \to a\langle z'', B, z\rangle \in P$ 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 aus dem Keller entfernt und das Suffix w_2' weglässt. 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 $\langle z', A, z\rangle \to \langle z'', B, z_1\rangle\langle z_1, C, z\rangle \in P$, gilt $\langle z', A, z\rangle \Rightarrow_G a\langle z'', B, z_1\rangle\langle z_1, C, z\rangle \Rightarrow_G^* aw_1'\langle z_1, C, z\rangle \Rightarrow_G^* aw_1'w_2' = w$. □

5.7.2.4. Kellerautomaten und kontextfreie Sprachen

Sätze 5.7.11 und 5.7.14 zeigen:

Theorem 5.7.15. *Kellerautomaten erkennen genau die kontextfreien Sprachen.*

Bemerkung 5.7.16. *Die zu Theorem 5.7.15 zugehörigen Beweise zeigen auch, dass PDAs mit einem Zustand auskommen, denn der in Satz 5.7.11 konstruierte PDA besitzt nur einen Zustand.*

Beispiel 5.7.17 (⋆). *Betrachte den PDA aus Beispiel 5.7.5* $M = (\{z_0, z_1\}, \{a, b\}, \{B, \#\}, \delta, z_0, \#)$ *mit*

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

und $\delta(z_i, a, A) = \emptyset$ *für alle anderen Fälle. Der Beweis von Satz 5.7.14 konstruiert die Grammatik* $G = (V, \Sigma, P, S)$ *mit*

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

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

- $\langle z_0, B, z_0\rangle \to a\langle z_0, B, z_1\rangle\langle z_1, B, z_0\rangle$*, 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 linker Seite gibt.*
- *Nun kann* $\langle z_0, B, z_0\rangle \to a\langle z_0, B, z_0\rangle\langle z_0, B, z_0\rangle$*, gelöscht werden, da sie die einzige Regel mit* $\langle z_0, B, z_0\rangle$ *als linker 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 linken Seite. Dies sind:* $\langle z_0, \#, z_0\rangle \to a\langle z_0, B, z_0\rangle\langle z_0, \#, z_0\rangle$, $\langle z_0, \#, z_1\rangle \to a\langle z_0, B, z_0\rangle\langle z_0, \#, z_1\rangle$ $\langle z_0, B, z_1\rangle \to a\langle z_0, B, z_0\rangle\langle z_0, B, z_1\rangle$,
- *Nun kann* $\langle z_0, \#, z_0\rangle \to a\langle z_0, B, z_1\rangle\langle z_1, \#, z_0\rangle$*, gelöscht werden, da es keine Produktion für* $\langle z_1, \#, z_0\rangle$ *gibt.*

Es verbleiben

$$\begin{aligned}\{&S \to \langle z_0, \#, z_0\rangle,\\ &S \to \langle z_0, \#, z_1\rangle,\\ &\langle z_0, B, z_1\rangle \to b,\\ &\langle z_1, B, z_1\rangle \to b,\\ &\langle z_0, \#, z_0\rangle \to \varepsilon,\\ &\langle z_1, \#, z_1\rangle \to \varepsilon,\\ &\langle z_0, \#, z_1\rangle \to a\langle z_0, B, z_1\rangle\langle z_1, \#, z_1\rangle,\\ &\langle z_0, B, z_1\rangle \to a\langle z_0, B, z_1\rangle\langle z_1, B, z_1\rangle\}\end{aligned}$$

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

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

Offensichtlich erzeugt B Wörter der From $a^{i+1}b^i$ und damit ist $L(G) = \{a^ib^i \mid i \in \mathbb{N}\}$. Beachte, dass bis auf $S \to \varepsilon$ ist G in Greibach-Normalform.

Sei $G' = (\{S, A, B, C\}, \{a, b\}, \{S \to aB, B \to b \mid aBC, C \to b\}, S)$, dann gilt $L(G') = L(G) \setminus \{\varepsilon\}$ und G' ist in Greibach-Normalform. Der Beweis von Satz 5.7.11 konstruiert für G den PDA $M = (\{z_0\}, \Sigma, \Sigma \cup V, \delta, z_0, S)$ mit

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

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

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

5.8. Deterministisch kontextfreie Sprachen

Für deterministische Kellerautomaten verwenden wir Automaten mit Endzuständen. Wir erlauben jedoch weiterhin ε-Übergänge. Damit der Übergang trotzdem deterministisch bleibt, darf ein ε-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 5.8.1 (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 DPDAs akzeptierten Sprachen heißen deterministisch kontextfrei*.*

Satz 5.8.2. *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 $\overline{w}$ verarbeitet. Im Anschluss daran wechselt er mit dem untersten Kellersymbol in den akzeptierenden Zustand z_2. Offensichtlich ist M deterministisch (d.h. ein DPDA). □

Satz 5.8.3. *Die Sprache $L = \{a^i b^i \mid i \in \mathbb{N}_{>0}\}$ 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). □

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

Theorem 5.8.4 (Eigenschaften deterministisch kontextfreier Sprachen)**.**

1. *Für deterministisch kontextfreie Sprachen gibt es eindeutige Grammatiken.*
2. *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 5.8.5. *Deterministisch kontextfreie Sprachen sind* nicht *abgeschlossen bezüglich Vereinigung und Schnitt.*

Beweis. Schnittbildung kann widerlegt werden durch die Schneiden der Sprachen $\{a^n b^n c^m \mid n, m \in \mathbb{N}_{>0}\}$ und $\{a^n b^m c^m \mid n, m \in \mathbb{N}_{>0}\}$, denn beide Sprachen sind deterministisch kontextfrei, ihr Schnitt hingegen ist die Sprache $\{a^n b^n c^n \mid n \in \mathbb{N}_{>0}\}$, die – wie bereits gezeigt, Satz 5.4.2 – 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. □

Der folgende Satz zeigt, dass das Schneiden mit einer regulären Sprache, die Menge der (deterministisch) kontextfreien Sprachen nicht verlässt:

Satz 5.8.6. *Der Schnitt einer (deterministisch) kontextfreien Sprachen mit einer regulären Sprache ist (deterministisch) kontextfrei.*

Beweis. Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$ ein PDA mit Endzuständen und $M' = (Z', \Sigma, \delta', z_0', E')$ ein DFA. Wir definieren den Kellerautomaten $M'' = (Z \times Z', \Sigma, \Gamma, \delta'', (z_0, z_0'), \#, E \times E')$ mit

- $((z_k, z_k'), B_1 \cdots B_m) \in \delta''((z_i, z_i'), a, A)$ falls $(z_k, B_1 \cdots B_m) \in \delta(z_i, a, A)$, $\delta'(z_i', a) = z_k'$ und
- $((z_k, z_i'), B_1 \cdots B_m) \in \delta''((z_i, z_i'), \varepsilon, A)$ falls $(z_k, B_1 \cdots B_m) \in \delta(z_i, \varepsilon, A)$.

Dann gilt $L(M'') = L(M) \cap L(M')$, denn M'' simuliert den PDA M und den DFA M' gleichzeitig, und akzeptiert nur dann, wenn beide Automaten akzeptieren. Schließlich lässt sich leicht prüfen, dass M'' ist deterministisch ist, wenn M deterministisch ist. □

5.9. ⋆ 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 betrachte 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 5.9.1. *Das Leerheitsproblem für kontextfreie Grammatiken ist entscheidbar.*

(⋆) Beweis. Sei L ein kontextfreie Sprache gegeben durch eine kontextfreie Grammatik. Wir können entsprechend Definition 3.2.4 prüfen, ob $\varepsilon \in L$ gilt. Ist dies der Fall, dann ist L nicht leer. Anderenfalls sei $G = (V, \Sigma, P, S)$ eine CFG in Chomsky-Normalform mit $L(G) = L$. Algorithmus 9 markiert alle Variablen $A \in V$ für die gilt $\{w \in \Sigma^* \mid A \Rightarrow_G^* w\} \neq \emptyset$.

Algorithmus 9 : ⋆ 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
  $W := \{A \in V \mid A \to a \in P, a \in \Sigma\}$;
  wiederhole
    $W_{alt} := W$;
    $W := W_{alt} \cup \{A \mid A \to BC \in P, B \in W_{alt}, C \in W_{alt}\}$;
  bis $W = W_{alt}$;
  return W
Ende

Nach Ausführung von Algorithmus 9 müssen wir nur testen, ob das Startsymbol S markiert wurde (d.h. ob $S \in W$ gilt). □

Satz 5.9.2. *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 $|L(G)| = \infty$ g.d.w. es ein Wort $z \in L(G)$ mit $n \leq |z| < 2n$ gibt:

- „$\Longleftarrow$“: Wenn es ein Wort $z \in L$ mit $|z| \geq n$ gibt, dann zeigt, dass Pumping-Lemma für CFGs (Lemma 5.4.1), dass alle Wörter $uv^iwx^iy \in L$ für $i \in \mathbb{N}$ liegen – das sind unendliche viele Wörter.
- „$\Longrightarrow$“: 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 $|z| \geq n$. 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. □

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.

5.9.1. Ein entscheidbares Problem

Ein entscheidbares Problem ist die Frage, ob eine deterministisch kontextfreie Sprache äquivalent zu einer regulären Sprache ist. Sei die deterministisch kontextfreie Sprache L_1 durch einen DPDA gegeben und die reguläre Sprache L_2 durch einen DFA. Dann prüfe, ob $\overline{L_1} \cap L_2 = \emptyset$ und $L_1 \cap \overline{L_2} = \emptyset$. Beides kann berechnet werden, da deterministische PDAs unter Komplementbildung abgeschlossen sind und DFAs ebenso, ein deterministischer PDA für den Schnitt konstruiert werden kann (siehe Satz 5.8.6) und das Leerheitsproblem für CFLs entscheidbar ist. Schließlich folgt aus $\overline{L_1} \cap L_2 = \emptyset$, dass $L_1 \subseteq L_2$ gilt, und aus $L_1 \cap \overline{L_2} = \emptyset$ folgt, dass $L_2 \subseteq L_1$ gilt. Damit kann $L_1 = L_2$ entschieden werden.

6. 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 $\ell \to r$ 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 $\alpha_1 A \alpha_2 \to \alpha_1 \alpha_3 \alpha_2$ haben müssen, wobei α_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.

6.1. ⋆ 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 6.1.1. *Eine Typ 1-Grammatik $G = (V, \Sigma, P, S)$ ist in* Kuroda-Normalform*, falls alle Produktionen in P einer der folgenden vier Formen entsprechen:*

$$A \to a \qquad A \to B \qquad A \to BC \qquad AB \to CD$$

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 $AB \to CD$.

Satz 6.1.2. *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 $A_a \to a$ und das Entfalten der Produktionen $A \to B_1 \cdots B_n$ 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 Wörter nicht.

Wir betrachten eine einzelne Ersetzung einer Regel $A_1 \cdots A_m \to B_1 \cdots B_n$ ($n \geq m+2$) 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 $i = 2, \ldots, n-1$ 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. □

6.2. 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 $\{a^i b^i c^i \mid i \in \mathbb{N}_{>0}\}$ 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 $\{a^i b^i c^i \mid i \in \mathbb{N}_{>0}\}$ 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. 6.1. 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 6.2.1 (Turingmaschine)**.** *Eine* Turingmaschine *(*TM*) ist ein 7-Tupel* $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ *wobei*

- Z *ist eine endliche Menge von* Zuständen,
- Σ *ist das (endliche)* Eingabealphabet,
- $\Gamma \supset \Sigma$ *ist das (endliche)* Bandalphabet,

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 */

$G := (V \cup \{A_a\}, \Sigma, \{\ell[A_a/a] \to r[A_a/a] \mid \ell \to r \in P\} \cup \{A_a \to a \mid a \in \Sigma\}, S)$;

Ende

/* Nun sind alle Regeln von der Form $A \to a$ oder $A_1 \cdots A_m \to B_1 \cdots B_n$ mit $A_i, B_j \in V$ */

für alle $A \to B_1 \cdots B_n \in P$ *mit* $n > 2$ **tue**

Seien $C_1, \ldots, C_{n-2}$ neue Variablen;

$V := V \cup \{C_1, \ldots, C_{n-2}\}$;

/* Ersetze in P die Produktion $A \to B_1 \cdots B_n$ durch neue Regeln */

$P := (P \setminus \{A \to B_1 \cdots B_n\})$
$\quad \cup \{A \to B_1 C_1\} \cup \{C_i \to B_{i+1} C_{i+1} \mid i = 1, \ldots, n-3\} \cup \{C_{n-2} \to B_{n-1} B_n\}$;

Ende

für alle $A_1 \cdots A_m \to B_1 \cdots B_n \in P$ *mit* $n \geq m + 2$ **tue**

Seien $D_2, \ldots, D_{n-1}$ neue Variablen;

$V := V \cup \{D_2, \ldots, D_{n-1}\}$;

/* Ersetze in P die Produktion $A_1 \cdots A_m \to B_1 \cdots B_n$ durch neue Regeln */

$P := (P \setminus \{A_1 \cdots A_m \to B_1 \cdots B_n\})$
$\quad \cup \{A_1 A_2 \to B_1 D_2\} \cup \{D_i A_{i+1} \to B_i D_{i+1} \mid i = 2, \ldots, m-1\}$
$\quad \cup \{D_i \to B_i D_{i+1} \mid i = m, \ldots, n-2\} \cup \{D_{n-1} \to B_{n-1} B_n\}$

Ende

für alle $A_1 \cdots A_n \to B_1 \cdots B_{n+1} \in P$ *mit* $n \geq 2$ **tue**

Seien $D_2, \ldots, D_n$ neue Variablen;

$V := V \cup \{D_2, \ldots, D_n\}$;

/* Ersetze in P die Produktion $A_1 \cdots A_n \to B_1 \cdots B_{n+1}$ durch neue Regeln */

$P := (P \setminus \{A_1 \cdots A_n \to B_1 \cdots B_n\}) \cup \{A_1 A_2 \to B_1 D_2\}$
$\quad \cup \{D_i A_{i+1} \to B_i D_{i+1} \mid i = 2, \ldots, n-1\} \cup \{D_n \to B_n B_{n+1}\}$

Ende

für alle $A_1 \cdots A_n \to B_1 \cdots B_n \in P$ *mit* $n > 2$ **tue**

Seien $D_2, \ldots, D_{n-1}$ neue Variablen;

$V := V \cup \{D_2, \ldots, D_{n-1}\}$;

/* Ersetze in P die Produktion $A_1 \cdots A_n \to B_1 \cdots B_n$ durch neue Regeln */

$P := (P \setminus \{A_1 \cdots A_n \to B_1 \cdots B_n\}) \cup \{A_1 A_2 \to B_1 D_2\}$
$\quad \cup \{D_i A_{i+1} \to B_i D_{i+1} \mid i = 2, \ldots, n-2\} \cup \{D_{n-1} A_n \to B_{n-1} B_n\}$

Ende

Gib die so entstandene Grammatik aus;

Ende

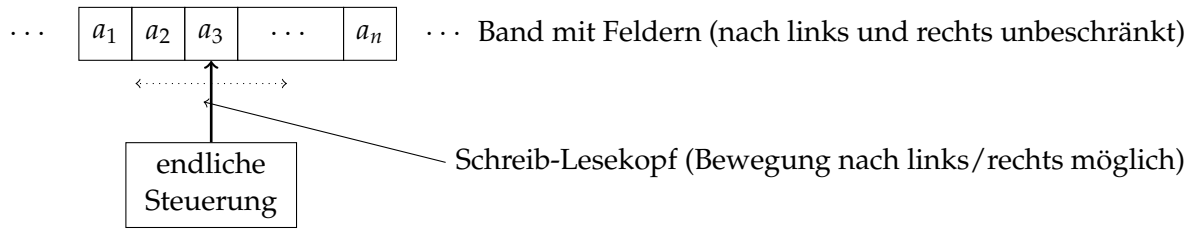


Abbildung 6.1.: Illustration der Turingmaschine

- *für eine deterministische TM ist* $\delta : (Z \setminus E) \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ *die* Zustandsüberführungsfunktion *(oder nur* Überführungsfunktion*) und für eine* nichtdeterministische *TM ist* $\delta : (Z \setminus E) \times \Gamma \to \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$ *die* Zustandsüberführungsfunktion *(oder nur* Überführungsfunktion*)*
- $z_0 \in Z$ *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. nichtdeterministische Turingmaschine.

Wir beschreiben den Zustandswechsel informell für die Überführungsfunktion δ und zunächst für die deterministische Turingmaschine: 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 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 der 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 6.2.2 (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 Turingmaschine 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[1]:

Definition 6.2.3 (Startkonfiguration einer TM)**.** *Für ein Eingabewort* w *ist die Startkonfiguration* $Start_M(w)$ *einer TM* $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ *das Wort* $z_0 w$*. Im Spezialfall* $w = \varepsilon$ *ist die Startkonfiguration* $z_0 \square$.

[1] Falls die Eingabe das leere Wort ε ist, steht der Schreib-Lesekopf auf einem Blank-Symbol

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

Definition 6.2.4 (Übergangsrelation 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* $\delta(z, a_1) = (z', c, N)$, $m \geq 0, n \geq 1, z \notin E$
- $b_1 \cdots b_m z a_1 \cdots a_n \vdash_M b_1 \cdots b_{m-1} z' b_m c a_2 \cdots a_n$, *wenn* $\delta(z, a_1) = (z', c, L)$, $m \geq 1, n \geq 1, z \notin E$
- $z a_1 \cdots a_n \vdash_M z' \square c a_2 \cdots a_n$, *wenn* $\delta(z, a_1) = (z', c, L), n \geq 1, z \notin E$
- $b_1 \cdots b_m z a_1 \cdots a_n \vdash_M b_1 \cdots b_m c z' a_2 \cdots a_n$, *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* $\delta(z, a_1) = (z', c, R), m \geq 0, z \notin E$

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 Nachfolgekonfigurationen, wenn der aktuelle Zustand ein Endzustand ist[2].

Die ersten drei Fälle von Definition 6.2.4 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 die Turingmaschine einen akzeptierenden Zustand erreicht (bzw. im nichtdeterministischen Fall: erreichen kann).

Definition 6.2.5 (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*

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

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:

[2] 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 δ als partielle Funktion sehen, die für die Fälle $\delta(q,a)$ mit $q \in E$ undefiniert ist. Alle Formalismen sind äquivalent und ineinander überführbar.

Beispiel 6.2.6. *Die Turingmaschine* $M = (\{z_0, z_1, z_2, z_3\}, \{0,1\}, \{0,1,\square\}, \delta, z_0, \square, \{z_3\})$ *mit*

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

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:*

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

6.3. 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, wird die Eingabe in spitzen Klammern gesetzt: Statt w ist die Eingabe nun $\langle w \rangle$. Eine Eingabe $a_1 \cdots a_m$ auf dem Band der Turingmaschine wird dementsprechend als $\langle a_1 \cdots a_m \rangle$ dargestellt und die Turingmaschine arbeitet auf dem Alphabet $\Sigma \supseteq \{\langle, \rangle\}$.

Definition 6.3.1. *Eine* linear beschränkt Turingmaschine *(linear bounded automaton, LBA) ist ein 8-Tupel* $M = (Z, \Sigma, \Gamma, \delta, z_0, \langle, \rangle, E)$*, wobei:*

- Z, Σ, Γ, δ, z_0 *und* E *sind wie bei NTMs*
- $\langle, \rangle \in \Sigma$ *sind Start- bzw. Endmarker*
- δ *überschreibt keinen der Marker*
- *bei* $\langle$ *gibt* δ *nie L aus und bei* $\rangle$ *gibt* δ *nie R aus.*

Die von M *akzeptierte Sprache* $L(M)$ *ist definiert als*

$$L(M) := \{w \in (\Sigma - \{\langle, \rangle\})^* \mid z_0 \langle w \rangle \vdash_M^* uzv \text{ für } u, v \in \Gamma^* \text{ und } z \in E\}$$

Beachte, dass LBAs nichtdeterministisch sind.

Im folgenden Abschnitt werden wir den Satz von Kuroda beweisen:

Theorem 6.3.2 (Satz von Kuroda)**.** *Kontextsensitive Sprachen werden genau von den LBAs erkannt.*

(⋆) Beweis. Dies wird durch Sätze 6.3.3 und 6.3.6 gezeigt. □

6.3.1. ⋆ Beweis des Satzes von Kuroda

Satz 6.3.3. *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 einen LBA, der als Bandalphabet Γ zumindest $(\Sigma \cup V \cup \{\square\}) \subseteq \Gamma$ verwendet. In den folgenden Erläuterung erwähnen wir die Start- und Endmarker $\langle, \rangle$ nicht explizit, gehen aber davon aus, dass der LBA entsprechend Rücksicht darauf nimmt und notwendige Ersetzungen macht bzw. definierte δ-Übergänge hat. Der LBA versucht (nichtdeterministisch) aus einem Wort $w \in \Sigma^*$ das Startsymbol S der Grammatik rückwärts herzuleiten, indem er Produktionen $\ell \to r \in P$ 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 ℓ. Für Produktionen $A \to a, A \to B, AB \to CD$ verändern diese Ersetzungen die Wortlänge nicht. Für den Fall $A \to BC$ 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 $\ell \to r$ 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, dass Symbol links vom Schreib-Lesekopf (d.h. für jedes Symbol in Σ 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).

Für Produktionen $A \to a$ und $A \to B$ 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 $AB \to CD$, wird B geschrieben und der Kopf nach links wechseln, dann A geschrieben und der nächste Schritt gestartet. Für Produktionen $A \to BC$ 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 wir beendet nachdem $\square$ mit einem anderen Symbol vertauscht wird.

Da für alle Produktionen $\ell \to r \in P$ gilt $|\ell| \leq |r|$ werden nur Teilworte r durch gleich lange 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 6.3.4. *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 des LBA 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 der LBA kommt mit linear beschränktem Platz aus (und kann daher konstruiert werden).*

Für Typ 0-Grammatiken kann es Produktionen der Form $\ell \to r$ 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 6.3.5. *Jede Typ i-Sprache (für i = 0, 1, 2, 3) wird von einer nichtdeterministischen Turingmaschine akzeptiert.*

Wir betrachten nun die andere Richtung:

Satz 6.3.6. *Sei M ein LBA. Dann ist $L(M)$ eine kontextsensitive Sprache.*

Beweis. Basierend auf dem LBA $M = (Z, \Sigma, \Gamma, \delta, z_0, \langle, \rangle, E)$ konstruieren wir eine kontextsensitive Grammatik G, sodass $L(G) = L(M)$ gilt. Die Idee für die Grammatik ist es, zunächst ein beliebiges Wort aus Σ^* zu generieren, dann den Ablauf des LBA auf diesem Wort mithilfe der Grammatik zu simulieren, und schließlich falls der LBA akzeptiert, das Wort endgültig herzustellen. Damit das Ganze gelingt, wird zunächst nicht nur das Wort aus Σ^*, 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) $\begin{bmatrix} u \\ v \end{bmatrix}$, wobei $u \in \Sigma$ und $v \in (\Gamma \cup Z)^+$. 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

$$P_0 := \{S \to \begin{bmatrix} a \\ z_0\langle a \end{bmatrix} A \mid a \in \Sigma\} \cup \{S \to \begin{bmatrix} a \\ z_0\langle a \rangle \end{bmatrix} \mid a \in \Sigma\}$$
$$\cup \{A \to \begin{bmatrix} a \\ a \end{bmatrix} A \mid a \in \Sigma\} \cup \{A \to \begin{bmatrix} a \\ a\rangle \end{bmatrix} \mid a \in \Sigma\}$$

Das leere Wort kann dadurch nicht erzeugt werden, dieses wird durch eine extra Regel direkt aus S erzeugt:

$$P_1 = \{S \to \varepsilon \mid \epsilon \in L(M)\}$$

Für alle anderen Wörter $w = a_1 \cdots a_n \in \Sigma^*$ gilt $S \Rightarrow^* \begin{bmatrix} a_1 \\ z_0\langle a_1 \end{bmatrix} \begin{bmatrix} a_2 \\ a_2 \end{bmatrix} \cdots \begin{bmatrix} a_n \\ a_n\rangle \end{bmatrix}$. Lesen wir die obere Reihe des erzeugten Wortes, ist dies w und die untere Reihe entspricht der Startkonfiguration $z_0\langle a_1 \cdots a_n\rangle$ von M für w.
Der nächste Regelsatz P_2 bildet die Übergangsrelation für M anhand der Definition von δ nach: Alle diese Regeln operieren auf den Tupelsequenzen $\begin{bmatrix} a_1 \\ v_1 \end{bmatrix} \cdots \begin{bmatrix} a_n \\ v_n \end{bmatrix}$ und lassen die obere Folge von Buchstaben unverändert (da sie nur die Konfiguration des LBA verändern).

$$P_2 := \left\{ \begin{bmatrix} d \\ za \end{bmatrix} \to \begin{bmatrix} d \\ z'b \end{bmatrix} \;\middle|\; (z', b, N) \in \delta(z, a), d \in \Sigma \right\}$$
$$\cup \left\{ \begin{bmatrix} d \\ c \end{bmatrix} \begin{bmatrix} e \\ za \end{bmatrix} \to \begin{bmatrix} d \\ z'c \end{bmatrix} \begin{bmatrix} e \\ b \end{bmatrix} \;\middle|\; (z', b, L) \in \delta(z, a), c \in \Gamma, d, e \in \Sigma \right\}$$
$$\cup \left\{ \begin{bmatrix} d \\ za \end{bmatrix} \begin{bmatrix} e \\ c \end{bmatrix} \to \begin{bmatrix} d \\ b \end{bmatrix} \begin{bmatrix} e \\ z'c \end{bmatrix} \;\middle|\; (z', b, R) \in \delta(z, a), c \in \Gamma, d, e \in \Sigma \right\}$$

Weitere Regeln sind notwendig um mit $\langle$ und $\rangle$ richtig umzugehen. Sie werden hier nicht angegeben.

Der dritte Regelsatz P_3 hat die Aufgabe nach Akzeptieren des LBA, aus den Tupelsequenzen das Wort $a_1 \cdots a_n$ zu erzeugen.

$$P_3 := \left\{ \begin{bmatrix} b \\ za \end{bmatrix} \to b \;\middle|\; z \in E, a \in \Gamma, b \in \Sigma \right\} \cup \left\{ \begin{bmatrix} b \\ a \end{bmatrix} \to b \;\middle|\; a \in \Gamma, b \in \Sigma \right\}$$

Auch hier sind weitere Regeln notwendig um mit $\langle$ und $\rangle$ richtig umzugehen.

Wenn $z \in E$ gilt $\begin{bmatrix} a_1 \\ b_1 \end{bmatrix} \cdots \begin{bmatrix} a_m \\ b_m \end{bmatrix} \begin{bmatrix} a_{m+1} \\ zb_{m+1} \end{bmatrix} \begin{bmatrix} a_{m+2} \\ b_{m+2} \end{bmatrix} \cdots \begin{bmatrix} a_n \\ b_n \end{bmatrix} \Rightarrow_{P_3}^* a_1 \cdots a_n$. Beachte, dass die Regeln aus P_3 auch in Nicht-Zielzuständen einzelne Tupel $\begin{bmatrix} a_i \\ b_i \end{bmatrix}$ in a_i umformen können, aber niemals ein Tupel $\begin{bmatrix} a_i \\ zb_i \end{bmatrix}$ mit $z \notin E$. Da die $\begin{bmatrix} a_i \\ zb_i \end{bmatrix}$ Variablen der Grammatik sind, lässt sich daher kein Wort aus Σ^* herleiten.

Schließlich sei $G = \left(\{S, A\} \cup \left\{ \begin{bmatrix} u \\ v \end{bmatrix} \mid u \in \Sigma, v \in (\Gamma \cup Z)^+ \right\}, \Sigma - \{\langle, \rangle\}, P_0 \cup P_1 \cup P_2 \cup P_3, S \right)$. Dann gilt für alle $w \in (\Sigma - \{\langle, \rangle\})^*$: $S \Rightarrow_G^* w$ g.d.w. $w \in L(M)$.

Des weiteren gilt, dass G eine kontextsensitive Grammatik ist, da es keine verkürzenden Regeln gibt. □

6.4. Turingmaschinen und Typ 0-Grammatiken

Satz 6.4.1. *Die durch (allgemeine) nichtdeterministische Turingmaschinen akzeptierten Sprachen sind genau die Typ 0-Sprachen.*

(⋆) Beweisskizze. Wir können die Konstruktionen aus dem Beweis von Satz 6.3.6 anpassen, um für jede nichtdeterministische Turingmaschine eine Typ 0-Grammatik zu definieren, sodass die erzeugte Sprache genau die von der NTM akzeptierte Sprache ist. Die Anpassungen sind u.a.: Zusätzliche Tupel $\begin{bmatrix} \$ \\ c \end{bmatrix}$ für \$ ein neues Symbol, das anzeigt, dass an dieser Stelle kein Wort mehr steht. Anschließend können Konfigurationen dargestellt werden, die länger als das Eingabewort sind, indem das Eingabewort mit \$-Symbolen aufgefüllt wird, d.h. die Satzformen für solche Konfiguration sind z.B. $\begin{bmatrix} a_1 \\ c_1 \end{bmatrix} \cdots \begin{bmatrix} a_n \\ c_n \end{bmatrix} \begin{bmatrix} \$ \\ c_{n+1} \end{bmatrix} \cdots \begin{bmatrix} \$ \\ z_i c_m \end{bmatrix} \begin{bmatrix} \$ \\ c_r \end{bmatrix}$, wenn $a_1 \cdots a_n$ das Eingabewort war und $c_1 \cdots c_{m-1} z_i c_m \cdots c_r$ die Turingmaschinenkonfiguration ist. Im Regelsatz P_3 müssen dann auch Regeln der Form $\begin{bmatrix} \$ \\ c_i \end{bmatrix} \to \varepsilon$ eingefügt werden, die es erlauben die unnötigen

Teile zu Löschen (diese Regeln sind dann vom Typ 0, aber nicht mehr kontextsensitiv). Diese Konstruktion und Satz 6.3.5 zeigen daher die Aussage. □

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.)

6.5. ⋆ LBA-Probleme

Für LBAs ist die Frage, ob *deterministische LBAs genau die gleichen Sprachen erkennen wie nichtdeterministische LBAs* bis heute ungeklärt (das sogenannte (erste) LBA-Problem). Das zweite LBA-Problem ist die Frage, ob die kontextsensitiven Sprachen unter Komplementbildung abgeschlossen sind. Dieses Problem wurde 1964 von Kuroda formuliert und 1988 sowohl von Neil Immerman als auch Róbert Szelepcsényi unabhängig gelöst und überraschenderweise positiv beantwortet:

Theorem 6.5.1 (Satz von Immerman und Szelepcsényi)**.** *Die kontextsensitiven Sprachen sind abgeschlossen unter Komplementbildung.*

Skizze. Sei $L \subseteq \Sigma^*$ eine kontextsensitive Sprache. Sei $G = (V, \Sigma, P, S)$ eine Typ 1-Grammatik mit $L(G) = L$. Wir konstruieren einen LBA, der $\overline{L} = \Sigma^* \setminus L$ akzeptiert.

Sei w ein Wort aus Σ^* und $n = |w|$. Der LBA berechnet zunächst die exakte Anzahl $A \in N$ der von S aus erzeugbaren Satzformen der Länge $\leq n$. Beachte, dass diese Zahl A nicht größer als $(|V| + |\Sigma| + 1)^n$ ist und daher in $(k+1) \cdot n$ Bits dargestellt werden kann, welche auf das Band der LBAs passen (indem Gruppen zu $k+1$-Bits in einem Symbol zusammengefasst werden).

Anschließend wird jede Satzform u der Länge $\leq n$ außer w selbst aufgezählt und jedes Mal nichtdeterministisch überprüft, ob $S \Rightarrow_G^* u$ gilt. Dabei wird ein Zähler geführt, der hochgezählt wird, falls $S \Rightarrow_G^* u$ gilt und gleich bleibt, falls die Ableitung nicht möglich ist. Wenn der Zähler die Zahl A erreicht, akzeptiert der LBA: Dann wurden alle aus S ableitbaren Wörter der Länge $\leq n$ gezählt, ohne dass w darunter war. Also liegt $w \notin L$ und damit $w \in \overline{L}$.

Es bleibt noch zu klären, wie die Zahl A berechnet wird. Sei A_m die Zahl der Satzformen, die in höchstens m Schritten aus S erzeugbar sind und deren Länge n nicht überschreitet (d.h. $A_m = |\{w \in (V \cup \Sigma)^* \mid |w| \leq n, S \Rightarrow^{\leq m} w\}|$). Wenn wir A_m berechnen können, können wir auch A berechnen indem wir mit $i = 0$ starten, i jeweils um 1 erhöhen und stets A_i berechnen, bis $A_{i+1} = A_i$ gilt (was passieren muss, da der Baum der möglichen Ableitungen von S zu einer Satzform der festen Länge n endlich verzweigend ist, sind die Pfade allesamt endlich).

Der Algorithmus zur Berechnung von A_m startet mit $A_0 = |\{S\}| = 1$ und berechnet *result* $= A_{m+1}$ durch Eingabe von A_m: Zunächst sei *result* $= 0$. Der Algorithmus zählt in einer äußeren Schleife alle Satzformen u bis zur Länge n auf und in einer inneren Schleife zählt er erneut alle Satzformen v bis zur Länge n auf. Vor Beginn der inneren Schleife wird ein Zähler *count* mit 0 initialisiert. In der inneren Schleife wird nichtdeterministisch überprüft, ob $S \Rightarrow^{\leq m} v$ gilt. Wenn ja, wird *count* um 1 erhöht. Es wird auch überprüft, ob $v = u$ oder $v \Rightarrow u$ gilt. Wenn ja, wird *result* um 1 erhöht und die Iteration von u fortgesetzt. Nach Ablauf der

inneren Schleife wird überprüft, ob $count = A_m$ gilt. Ist dies nicht der Fall, wird diese nichtdeterministische Berechnung verworfen. Anderenfalls war die Maschine auf einem richtigen nichtdeterministischen Pfad und hat in der inneren Schleife für alle in $\leq m$ Schritten aus S herleitbaren Satzformen v (das sind A_m viele) geprüft, ob durch Verlängern mit $=$ oder $\Rightarrow$ (d.h. in $\leq m+1$ Schritten) eine der Satzformen u herleitbar ist. d.h. *result* enthält den Wert A_{m+1}. □

7. Zusammenfassung und Überblick

Wir haben in diesem Teil die formalen Sprachen und ihre Klassifizierung in Form der Chomsky-Hierarchie kennengelernt. 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) (nichtdeterministisch)	
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 $\{a^n b^n \mid n \in \mathbb{N}\}$ ist vom Typ 2 aber nicht vom Typ 3.
- Die Sprache $\{w \in \{a,b\}^* \mid w \text{ ist ein Palindrom}\}$ ist vom Typ 2 aber nicht deterministisch-kontextfrei.
- Die Sprache $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ ist vom 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 vom 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 Entscheidbarkeitsprobleme:

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

Teil II.

Berechenbarkeitstheorie

8. Der intuitive Berechenbarkeitsbegriff

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 formalen 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 8.0.1 (Berechenbarkeit)**.** *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ 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 $(n_1, \ldots, n_k) \in \mathbb{N}^k$ startet, dann stoppt der Algorithmus nach endlichen vielen Berechnungsschritten und liefert den Wert von $f(n_1, \ldots, n_k)$ als Ergebnis. Falls f eine partielle Funktion ist (d.h. nicht für alle Eingaben definiert ist), dann soll der Algorithmus bei einer Eingabe $(n_1, \ldots, n_k)$ für die $f(n_1, \ldots, n_k)$ undefiniert ist, nicht anhalten, sondern unendlich lange laufen.*

Beispiel 8.0.2. *Der Algorithmus*

Eingabe : *Zahl $n \in \mathbb{N}$*
Beginn
  solange `true` **tue**
    skip
  Ende
Ende

berechnet die partielle Funktion $f_1 : \mathbb{N} \to \mathbb{N}$, die für alle Eingaben undefiniert ist (man schreibt dies auch als $f_1(x) = \perp$).

Der Algorithmus

Eingabe : *Zahlen $n_1, n_2 \in \mathbb{N}$*
Beginn
  hilf := $n_1 + n_2$;
  return *hilf*
Ende

berechnet die Funktion $f_2 : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$, welche die Summe der beiden Eingabezahlen berechnet (d.h. $f_2(x, y) = x + y$).

Beispiel 8.0.3. *Betrachte die Funktion*

$$f_3(n) = \begin{cases} 1, & \text{falls } n \text{ ein Präfix der Ziffern der Dezimalzahldarstellung von } \pi \text{ ist} \\ 0, & \text{sonst} \end{cases}$$

Z.B. gilt $f_3(31) = 1$ und $f_3(314) = 1$, aber $f_3(2) = 0$ und $f_3(315) = 0$. Die Funktion f_3 ist berechenbar, da man mit Näherungsverfahren eine beliebige (aber feste) Anzahl der Dezimalzahldarstellung von π genau berechnen kann und im Anschluss nur noch vergleichen muss.

Beispiel 8.0.4. *Für die Funktion*

$$f_4(n) = \begin{cases} 1, & \text{falls } n \text{ ein Teilwort der Ziffern der Dezimalzahldarstellung von } \pi \text{ ist} \\ 0, & \text{sonst} \end{cases}$$

ist es erst einmal unklar, ob sie berechenbar ist. Wenn wir alle Teilworte von π ausprobieren müssen, können wir vermutlich keinen (anhaltenden) Algorithmus angeben.

Wenn wir jedoch wüssten, dass jede Zahl n irgendwann in π als Teilwort vorkommen muss (weil die Ziffernfolge von π derart zufällig ist, dass sie jede Folge irgendwann erzeugt), dann wäre g auch berechenbar, denn es würde gelten $f_4(n) = 1$ für alle $n \in \mathbb{N}$.

Beispiel 8.0.5. *Für die Funktion*

$$f_5(n) = \begin{cases} 1, & \text{falls die Dezimaldarstellung von } \pi \text{ das Wort } 3^n \text{ als Teilwort besitzt} \\ 0, & \text{sonst} \end{cases}$$

sieht die Frage der Berechenbarkeit zunächst genauso schwierig aus, wie für die Funktion f_4. Allerdings ist dem nicht so, denn entweder gibt es für jedes n das Teilwort 3^n in der Dezimaldarstellung von π und $f_5(n) = 1$ für alle n, oder es gibt eine Zahl n_0, sodass 3^{n_0} noch als Teilwort vorkommt, aber die Folgen 3^n für $n > n_0$ kommen alle nicht in der Dezimaldarstellung von π vor. Dann ist $f_5(n) = 1$ für $n \geq n_0$ und $f_5(n) = 0$ sonst.

Welcher der beiden Fälle nun zutrifft, wissen wir nicht. Aber einer der beiden Fälle muss gelten und in beiden Fällen können wir einen Algorithmus angeben, der f_5 berechnet.

Daher ist f_5 berechenbar.

Das letzte Beispiel zeigt, dass wir für das Entscheiden der Berechenbarkeit einer Funktion f nicht unbedingt den geforderten Algorithmus, der f berechnet, angeben müssen. Wir müssen nur zeigen, dass dieser *existiert* (bzw. nicht existieren kann). Daher ist unsere Definition der Berechenbarkeit nicht *konstruktiv* und liefert nicht gleich den Algorithmus zur Berechnung.

Beispiel 8.0.6. *Die Funktion*

$$f_6(n) = \begin{cases} 1, & \text{falls deterministische LBAs genau die gleichen Sprachen erkennen} \\ & \text{wie nichtdeterministische LBAs} \\ 0, & \text{sonst} \end{cases}$$

ist berechenbar, denn entweder hat das erste LBA-Problem eine positive Lösung und $f_6(n) = 1$ für alle $n \in \mathbb{N}$, oder das LBA-Problem hat eine negative Lösung und $f_6(n) = 0$ für alle $n \in \mathbb{N}$. Daher existiert ein Algorithmus der f_6 berechnet (wir wissen nur nicht, welcher von beiden der richtige ist).

Sei f^r die Funktion

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

Wir hatten in Beispiel 8.0.3 gesehen, dass die Funktion f^π berechenbar ist (dort hieß diese Funktion f_3). Nun könnte man vermuten, dass jede Funktion f^r für eine reelle Zahl r berechenbar ist. Allerdings ist dem nicht so, denn es gibt überabzählbar viele reelle Zahlen aber nur abzählbar viele Algorithmen (bzw. Programme einer Programmiersprache). Für verschiedene reelle Zahlen $r_1 \neq r_2$ benötigen wir auch verschiedene Algorithmen zur Berechnung von f^{r_1} und f^{r_2} (denn es muss einen Präfix n geben, sodass $f^{r_1}(n) \neq f^{r_2}(n)$). Daraus folgt, dass wir eine überabzählbare Anzahl an Algorithmen bräuchten, die uns aber nicht zur Verfügung stehen. D.h. es muss reelle Zahlen r geben, sodass f^r nicht berechenbar ist.

Im Folgenden werden wir verschiedene formale Definitionen (und Modelle) zur Berechenbarkeit sehen. Einige davon wurden von Alan Turing und Alonzo Church in den 1930er vorgeschlagen. Neben Berechenbarkeit anhand von Turingmaschinen (d.h. Turing-Berechenbarkeit) lernen wir WHILE-Programme, GOTO-Programme, μ-rekursive Funktionen und die darauf basierenden Berechenbarkeitsbegriffe kennen. Es hat sich (erstaunlicherweise) gezeigt, dass alle 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 (äquivalent WHILE-berechenbaren, GOTO-berechenbaren, μ-rekursiven) Funktionen stimmt genau mit der Klasse der intuitiv berechenbaren Funktionen überein.

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

9. Turings Modell der Berechenbarkeit

Wir haben die von Alan Turing vorgeschlagenen Turingmaschinen bereits in Kapitel 6 kennengelernt. In diesem Abschnitt definieren wir zunächst darauf aufbauend den Begriff der Turingberechenbarkeit einer Funktion. Im Anschluss betrachten wir Varianten von Turingmaschinen und zeigen schließlich, wie man größere Turingmaschinen durch Komposition von kleineren Turingmaschinen konstruieren kann.

9.1. Turingmaschinen und Turingberechenbarkeit

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, Σ 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 δ die Zustandsüberführungsfunktion (mit $\delta : (Z \setminus E) \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ für eine DTM und $\delta : (Z \setminus E) \times \Gamma \to \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$ 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 6.2.5). 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 9.1.1. *Sei $bin(n)$ die Binärdarstellung von $n \in \mathbb{N}$. Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt* turingberechenbar, *falls es eine (deterministische) Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ gibt, sodass für alle $n_1, \ldots, n_k, m \in \mathbb{N}$ gilt:*

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

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

$$Start_M(u) \vdash^* \square \cdots \square z_e v \square \cdots \square \text{ mit } z_e \in E.$$

Eine Konsequenz (für beide Definitionen der Turingberechenbarkeit) ist: Falls $f(n_1, \ldots, n_k)$ bzw. $f(u)$ undefiniert ist, dann kann die Maschine ewig laufen.

Beispiel 9.1.2. *Die Nachfolgerfunktion $f(x) = x + 1$ für alle $x \in \mathbb{N}$ ist turingberechenbar. Wir haben eine entsprechende Turingmaschine bereits in Beispiel 6.2.6 angegeben.*

Beispiel 9.1.3. *Die Identitätsfunktion $f(x) = x$ für alle $x \in \mathbb{N}$ 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: $z_0 bin(n) \vdash^* z_0 bin(n)$ für alle $n \in \mathbb{N}$.*

Die überall undefinierte Funktion Ω 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).

9.2. Mehrspuren- und Mehrband-Turingmaschinen

Unsere bisher eingeführten Turingmaschinen haben ein Band, dass 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- und 1-Spur-Turingmaschinen zurückführen lassen. Dies zeigt, dass die Varianten den Berechenbarkeitsbegriff unverändert lassen.

9.2.1. Mehrspuren-Turingmaschinen

Definition 9.2.1 (Mehrspuren-Turingmaschine)**.** *Für $k \in \mathbb{N}_{>0}$ 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*,*
- *Σ ist das (endliche)* Eingabealphabet*,*
- *$\Gamma \supset \Sigma$ ist das (endliche)* Bandalphabet*,*
- *für eine deterministische TM ist $\delta : (Z \setminus E) \times \Gamma^k \to Z \times \Gamma^k \times \{L, R, N\}$ die* Zustandsüberführungsfunktion *und für eine* nichtdeterministische *TM ist $\delta : (Z \setminus E) \times \Gamma^k \to \mathcal{P}(Z \times \Gamma^k \times \{L, R, N\})$ die* Zustandsüberführungsfunktion*,*
- *$z_0 \in Z$ 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-Spur-TM) ist die Idee der k-Spuren-TM. dass das Band der Turingmaschine in k übereinander liegende Spuren geteilt ist. Abb. 9.1 illustriert diese Idee:

Die Turingmaschine liest ein Tupel $(a_1, \ldots, a_k)$ 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 9.2.2. *Jede Mehrspuren-Turingmaschine kann auf einer 1-Spur-Turingmaschine simuliert werden.*

Beweis. Verwende als Bandalphabet $\Gamma \cup \Gamma^k$, als Eingabealphabet Σ und als Blank-Symbol $\square$. Beginnend mit einer Eingabe aus Σ^*, kann die Mehrspurendarstellung erzeugt werden, indem jedes Symbol $a \in \Sigma$ aus der Eingabe durch das k-Tupel $(a, \square, \ldots, \square)$ ersetzt wird. Anschließend kann jeder Berechnungsschritt der Mehrspurenmaschine auf der 1-Spur-Maschine simuliert

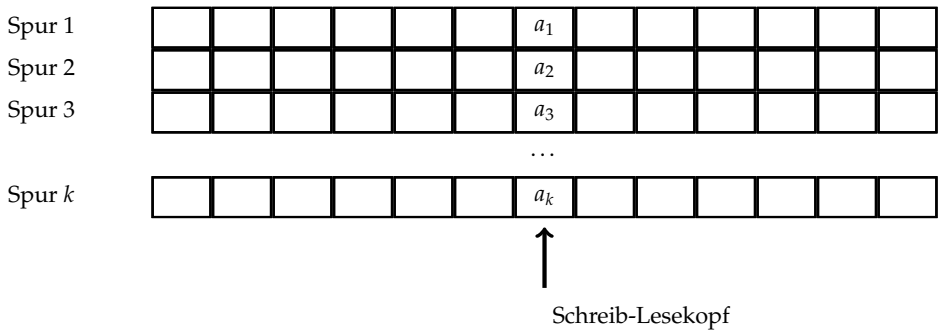


Abbildung 9.1.: Mehrspuren-Turingmaschine

werden und schließlich die Ausgabe erzeugt werden, indem jeder Eintrag $(a_1, \ldots, a_k)$ durch a_1 ersetzt wird. Die Akzeptanz der Simulation ist dieselbe und auch die Turingberechenbarkeit bleibt dieselbe. □

Beispiel 9.2.3. *Das folgende Beispiel ist in ähnlicher Weise in (HMU06) zu finden. Wir programmieren eine Turingmaschine, welche die Sprache $\{wcw \mid w \in \{a,b\}^+\}$ erkennt. Dafür verwenden wir 2 Spuren und verzichten auf das Erstellen und Entfernen der 2-Spuren-Darstellung 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 wcw auf Spur 1 wird nur gelesen, aber nicht verändert.*
- *Auf der zweiten Spur wird nur das Blank-Symbol und das Symbol ✓ 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 $Z = \{z_0, z_{1a}, z_{1b}, z_{2a}, z_{2b}, z_3, \ldots, z_9\}$. 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 δ 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 dass 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*.*
- $\delta(z_i, (s, t)) = (z_9, (s, t), N)$ *mit* $i \neq 8$ *für alle anderen Fälle verbleibe oder wechsele in den verwerfenden Zustand.*

Ein Lauf der Turingmaschine für das Wort abcab ist:

$$
\begin{array}{l}
z_0 \begin{array}{ccccc} a & b & c & a & b \\ \square & \square & \square & \square & \square \end{array} \vdash \begin{array}{ccccc} a & & b & c & a & b \\ \checkmark & z_{1a} & \square & \square & \square & \square \end{array} \vdash \begin{array}{ccccc} a & b & & c & a & b \\ \checkmark & \square & z_{1a} & \square & \square & \square \end{array} \vdash \begin{array}{ccccc} a & b & c & & a & b \\ \checkmark & \square & \square & z_{2a} & \square & \square \end{array} \\[2ex]
\vdash \begin{array}{ccccc} a & b & & c & a & b \\ \checkmark & \square & z_3 & \square & \checkmark & \square \end{array} \vdash \begin{array}{ccccc} a & & b & c & a & b \\ \checkmark & z_4 & \square & \square & \checkmark & \square \end{array} \vdash \begin{array}{ccccc} & a & b & c & a & b \\ z_5 & \checkmark & \square & \square & \checkmark & \square \end{array} \vdash \begin{array}{ccccc} a & & b & c & a & b \\ \checkmark & z_0 & \square & \square & \checkmark & \square \end{array} \\[2ex]
\vdash \begin{array}{ccccc} a & b & & c & a & b \\ \checkmark & \checkmark & z_{1b} & \square & \checkmark & \square \end{array} \vdash \begin{array}{ccccc} a & b & c & & a & b \\ \checkmark & \checkmark & \square & z_{2b} & \checkmark & \square \end{array} \vdash \begin{array}{ccccc} a & b & c & a & & b \\ \checkmark & \checkmark & \square & \checkmark & z_{2b} & \square \end{array} \vdash \begin{array}{ccccc} a & b & c & & a & b \\ \checkmark & \checkmark & \square & z_3 & \checkmark & \checkmark \end{array} \\[2ex]
\vdash \begin{array}{ccccc} a & b & & c & a & b \\ \checkmark & \checkmark & z_3 & \square & \checkmark & \checkmark \end{array} \vdash \begin{array}{ccccc} a & & b & c & a & b \\ \checkmark & z_4 & \checkmark & \square & \checkmark & \checkmark \end{array} \vdash \begin{array}{ccccc} a & b & & c & a & b \\ \checkmark & \checkmark & z_6 & \square & \checkmark & \checkmark \end{array} \vdash \begin{array}{ccccc} a & b & c & & a & b \\ \checkmark & \checkmark & \square & z_7 & \checkmark & \checkmark \end{array} \\[2ex]
\vdash \begin{array}{ccccc} a & b & c & a & & b \\ \checkmark & \checkmark & \square & \checkmark & z_7 & \checkmark \end{array} \vdash \begin{array}{ccccc} a & b & c & a & b & \square \\ \checkmark & \checkmark & \square & \checkmark & \checkmark & \square \end{array} \vdash \begin{array}{ccccc} a & b & c & a & b & \square \\ \checkmark & \checkmark & \square & \checkmark & \checkmark & \square \end{array}
\end{array}
$$

9.2.2. 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. 9.2 zu finden.

Definition 9.2.4 (Mehrband-Turingmaschine)**.** *Für* $k \in \mathbb{N}_{>0}$ *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,
- Σ *ist das (endliche)* Eingabealphabet,
- $\Gamma \supset \Sigma$ *ist das (endliche)* Bandalphabet,

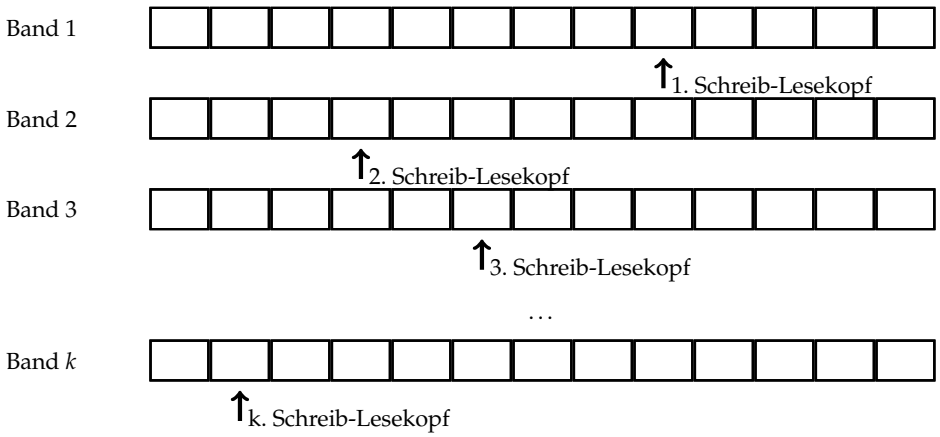


Abbildung 9.2.: Illustration der Mehrband-Turingmaschine

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

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 9.2.5. *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 $2k$-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. 9.3 illustriert diese Darstellung.

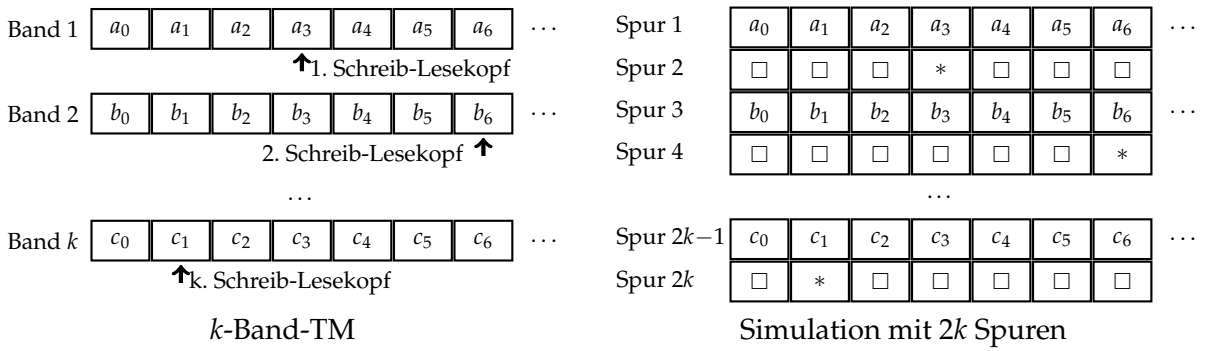


Abbildung 9.3.: Simulation der Turingmaschine mit k-Bändern

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 Zustände) die Bandinhalte an den 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. □

9.3. ⋆ Modulare Konstruktion von Turingmaschinen

In diesem Abschnitt verwenden wir sowohl 1-Band-Turingmaschinen als auch die eingeführten Mehrband-Turingmaschinen, um einfache Rechenoperationen auf diesen zu „programmieren“. Insbesondere verwenden wir die Komposition von Turingmaschinen, um aus Turingmaschinen für ganz einfache Operationen, Turingmaschinen für komplexere Operationen zu konstruieren.

9.3.1. Notation

Wir verwenden dieselbe Notation wie (Sch08), um bestimmte Turingmaschinen zu bezeichnen. Wenn M eine 1-Band-Turingmaschine ist, dann schreiben wir $M(i,k)$ für die k-Band-Turingmaschine (mit $i \leq k$), welche die Operationen von M auf dem i-ten Band durchführt und alle anderen Bänder unverändert lässt. Wenn k nicht von Bedeutung ist (aber in jedem Fall groß genug gewählt wurde bzw. gewählt werden kann), schreiben wir statt $M(i,k)$ oft einfach $M(i)$.

9.3.2. Einfache Rechenoperationen

In Beispiel 6.2.6 haben wir eine Turingmaschine angegeben, die zur Eingabe (einer Zahl in Binärdarstellung) 1 dazu addiert und das Ergebnis auf dem Band zu Verfügung stellt. Wir bezeichnen diese Turingmaschine ab sofort mit „Band := Band+1“. Entsprechend der vorher eingeführten Notation bezeichnet dann „Band := Band+1“(i) die k-Band-Turingmaschine ($k \geq i$), welche die Nachfolgerfunktion für den Wert auf Band i berechnet. Wir schreiben dies kürzer als „Band i := Band $i+1$“. Wir führen weitere Bezeichnungen für Turingmaschinen an, die allesamt einfach zu konstruieren sind (wir geben die konkreten Maschinen nicht an und lassen deren Konstruktion als Übungsaufgabe).

Mit „Band i := Band $i-1$“ bezeichnen wir die k-Band-Turingmaschine ($k \geq i$), die eine (angepasste) Subtraktion von 1 auf Band i durchführt, wobei die Anpassung ist, dass wir $0-1=0$ definieren. Mit „Band i := 0“ bezeichnen wir die Turingmaschine, welche den Inhalt auf Band i mit der Zahl 0 (jeweils in Binärdarstellung) überschreibt. Mit „Band i := Band j“ bezeichnen wir die k-Band-Turingmaschine ($k \geq i$ und $k \geq j$), welche die Zahl von Band j auf Band i kopiert (und den Inhalt von Band i dabei überschreibt).

9.3.3. Komposition von Turingmaschinen

Seien $M_1 = (Z_1, \Sigma, \Gamma_1, \delta_1, z_1, \square, E_1)$ und $M_2 = (Z_1, \Sigma, \Gamma_2, \delta_2, z_2, \square, E_2)$ k-Band-Turingmaschinen. Mit $M_1; M_2$ bezeichnen wir jene Turingmaschine, welche M_1 und M_2 nacheinander ausführt: Seien Z_1 und Z_2 paarweise disjunkt[1]. Dann ist

$$M_1; M_2 = (Z_1 \cup Z_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, z_1, \square, E_2)$$

mit

$$\delta(z, (a_1, \ldots, a_k)) = \begin{cases} \delta_1(z, (a_1, \ldots, a_k)) & \text{falls } z \in Z_1 \setminus E_1 \text{ und } (a_1, \ldots, a_k) \in \Gamma_1^k \\ \delta_2(z, (a_1, \ldots, a_k)) & \text{falls } z \in Z_2 \setminus E_2 \text{ und } (a_1, \ldots, a_k) \in \Gamma_2^k \\ (z_2, (a_1, \ldots, a_k), N^k) & \text{falls } z \in E_1 \text{ und } (a_1, \ldots, a_k) \in \Gamma_1^k \end{cases}$$

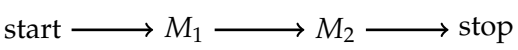
Die Maschine $M_1; M_2$ führt erst M_1 aus, und – falls diese in einem Endzustand $z \in E_1$ ist, wechselt sie in den Startzustand von M_2 um diese anschließend auszuführen.

Mithilfe der Hintereinanderschaltung können wir z.B. die Turingmaschine

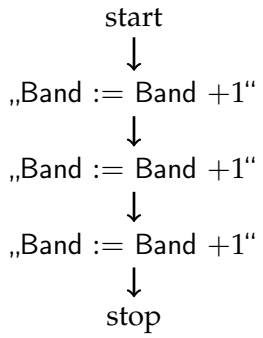
„Band := Band+3“ als „Band := Band+1“;„Band := Band+1“;„Band := Band+1“

konstruieren.

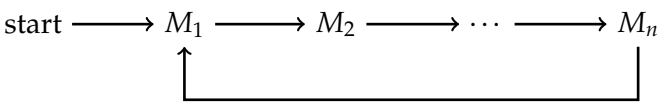
Wir verwenden (analog zu (Sch08)) folgendes Flussdiagramm, um die Hintereinanderschaltung $M_1; M_2$ zu notieren:



Die „Band := Band+3“-Maschine kann entsprechend durch das Flussdiagramm



repräsentiert werden. Beachte, dass die Konstruktion der Hintereinanderschaltung es auch ermöglicht, die Verkettung zyklisch zu konstruieren. Z.B. kann die eingeführte Konstruktion leicht angepasst werden, um für Turingmaschinen $M_1, \ldots, M_n$ den folgenden Zyklus zu konstruieren:

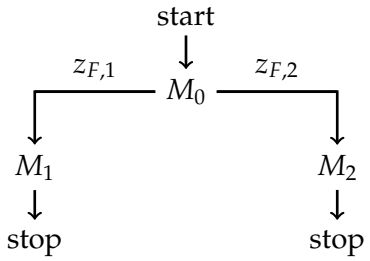


[1] Anderenfalls benenne eine der Zustandsmengen und Maschinen konsistent mit frischen Namen um.

Zunächst scheint diese Konstruktion wenig sinnvoll, da (wie das Flussdiagramm schon zeigt) die konstruierte Maschine nicht anhält.

Aber wir werden diese bald zusammen mit der Verzweigung verwenden, um den Zyklus nur in manchen Fällen zu verwenden und damit Schleifen zu repräsentieren.

Seien M_0, M_1, M_2 Turingmaschinen. Dann wird eine „verzweigende Fortsetzung“ je nach Ergebnis des Laufs der Turingmaschine M_0 durch das folgende Flussdiagramm dargestellt, wobei $z_{F,1}$ und $z_{F,2}$ die Endzustände von M_0 seien:



Die Konstruktion dazu fügt entsprechende Übergänge

$$\delta(z_{F,1}, (a_1, \ldots, a_k)) = (z_{0,M_1}, (a_1, \ldots, a_k), N) \text{ und } \delta(z_{F,2}, (a_1, \ldots, a_k)) = (z_{0,M_2}, (a_1, \ldots, a_k), N)$$

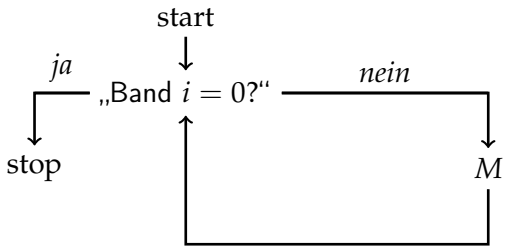
ein, wobei δ die neue Übergangsfunktion ist und z_{0,M_i} den Startzustand von M_i (für $i = 1, 2$) bezeichnet.

Ein Beispiel für die Turingmaschine M_0 in der vorherigen Konstruktion ist die Turingmaschine die Zustände $\{z_0, z_1, ja, nein\}$ hat und die Übergangsfunktion

$$\begin{array}{lcll} \delta(z_0, a) & = & (nein, a, N) & \text{für } a \neq 0 \\ \delta(z_0, 0) & = & (z_1, 0, R) & \\ \delta(z_1, a) & = & (nein, a, L) & \text{für } a \neq \square \\ \delta(z_1, \square) & = & (ja, \square, L) & \end{array}$$

verwendet, wobei *ja* und *nein* Endzustände sind und z_0 der Startzustand ist. Die TM prüft, ob das Band eine 0 enthält oder nicht. Dementsprechend nennen wir die Maschine „Band=0?“ und schreiben „Band $i = 0$?“ anstelle von „Band$=0$?“(i).

Verwenden wir die Verzweigung zusammen mit der „Band i=0?“-Maschine und dem Bilden von Zyklen, so können wir für eine beliebige Turingmaschine M eine Turingmaschine mit folgendem Flussdiagramm konstruieren:



Die Turingmaschine M wird solange immer wieder aufgerufen, bis das i-te Band die Zahl 0 enthält. Die Maschine nennen wir nun „WHILE Band $i \neq 0$ DO M“.

Diese Ausführungen zeigen schon, dass wir Programme einer einfachen imperativen Programmiersprache mit Zuweisungen, Verzweigungen und Schleifen durch Turingmaschinen simulieren können. Im nächsten Kapitel werden wir diese Konstruktionen verwenden, um drei grundsätzliche imperative Programmiersprachen bezüglich der induzierten Berechenbarkeitsbegriffe auch im Vergleich zur Turingberechenbarkeit zu untersuchen.

10. ⋆ LOOP-, WHILE- und GOTO-Berechenbarkeit

In diesem Kapitel betrachten wir drei einfache imperative Programmiersprachen: LOOP-Programme, WHILE-Programme und GOTO-Programme. Für alle drei Sprachen definieren wir neben ihrer Syntax auch die operationale Semantik, d.h. wie Programme der Sprache ausgeführt werden. Im Anschluss zeigen wir, dass WHILE- und GOTO-Programme den selben Berechenbarkeitsbegriff induzieren, der äquivalent zur Turingberechenbarkeit ist, während LOOP-Programme einen schwächeren Berechenbarkeitsbegriff induzieren (d.h. nicht jede turingberechenbare Funktion ist auch LOOP-berechenbar).

10.1. LOOP-Programme

10.1.1. Syntax von LOOP-Programmen

Die Syntax von LOOP-Programmen kann durch die folgende kontextfreie Grammatik beschrieben werden. Sie werden durch die CFG (V, Σ, P, Prg) erzeugt, wobei:

$$
\begin{array}{lcll}
V & = & \{Prg, Var, Id, Const\} & \\
\Sigma & = & \{\textbf{LOOP}, \textbf{DO}, \textbf{END}, x, 0, \ldots, 9, ;, :=, +, -\} & \\
P & = & \{Prg & \to \ \textbf{LOOP}\ Var\ \textbf{DO}\ Prg\ \textbf{END} \\
 & & & \mid Prg; Prg \\
 & & & \mid Var := Var + Const \\
 & & & \mid Var := Var - Const \\
 & & Var & \to \ x_{Id} \\
 & & Const & \to \ Id \\
 & & Id & \to \ 0 \mid 1 \mid \ldots \mid 9 \mid 1Id \mid 2Id \mid \ldots \mid 9Id\}
\end{array}
$$

Daher erzeugt das Nichtterminal *Var* die Wörter $x_0, x_1, x_2, \ldots$, d.h. einen Variablennamen aus einer abzählbar unendlichen Menge von Namen. Das Nichtterminal *Const* erzeugt die Konstanten der LOOP-Programme (alle natürlichen Zahlen).

Programme können durch das LOOP-Konstrukt der Form **LOOP** x_i **DO** P **END** (wobei P ein Programm ist), durch Sequenzen $P_1; P_2$ oder durch Zuweisungen $x_i := x_j + c$ bzw. $x_i := x_j - c$, wobei c eine Konstante ist, gebildet werden.

10.1.2. Semantik von LOOP-Programmen

Zur Modellierung des Speicherzustandes, d.h. der Belegung der Variablen mit Werten, definieren Variablenbelegungen:

Definition 10.1.1 (Variablenbelegung)**.** *Eine* Variablenbelegung ρ *ist eine endliche Abbildung mit Einträgen* $x_i \mapsto n$ *mit* x_i *ist Variable und* $n \in \mathbb{N}$*. Wir definieren die Anwendung einer Variablenbele-*

gung ρ auf eine Variable x_i als

$$\rho(x_i) := \begin{cases} n, & \text{wenn } x_i \mapsto n \in \rho \\ 0, & \text{sonst} \end{cases}$$

Wir schreiben $\rho\{x_i \mapsto m\}$ für die Abbildung, die an der Stelle für x_i so modifiziert wurde, dass sie x_i auf m abbildet und sich ansonsten wie ρ verhält, d.h.

$$\rho\{x_i \mapsto m\}(x_j) = \begin{cases} \rho(x_j), & \text{wenn } x_j \neq x_i \\ m, & \text{wenn } x_j = x_i \end{cases}$$

Die Semantik von LOOP-Programmen kann als Abfolge von Berechnungsschritten $\xrightarrow[\text{LOOP}]{}$ auf Paaren (ρ, P) beschrieben werden, wobei ρ eine Variablenbelegung und P ein LOOP-Programm oder ein „leeres“ Programm ε ist.

Definition 10.1.2 (Berechnungsschritt $\xrightarrow[\text{LOOP}]{}$)**.** *Die Definition eines Berechnungsschrittes für eine Variablenbelegung ρ und ein LOOP-Programm ist:*

- $(\rho, x_i := x_j + c) \xrightarrow[LOOP]{} (\rho\{x_i \mapsto n\}, \varepsilon)$ *wobei* $n = \rho(x_j) + c$
- $(\rho, x_i := x_j - c) \xrightarrow[LOOP]{} (\rho\{x_i \mapsto n\}, \varepsilon)$ *wobei* $n = \max(0, \rho(x_j) - c)$
- $(\rho, P_1; P_2) \xrightarrow[LOOP]{} (\rho', P_1'; P_2)$ *wenn* $(\rho, P_1) \xrightarrow[LOOP]{} (\rho', P_1')$ *und* $P_1' \neq \varepsilon$
- $(\rho, P_1; P_2) \xrightarrow[LOOP]{} (\rho', P_2)$ *wenn* $(\rho, P_1) \xrightarrow[LOOP]{} (\rho', \varepsilon)$
- $(\rho, \textbf{LOOP } x_i \textbf{ DO } P \textbf{ END}) \xrightarrow[LOOP]{} (\rho, \underbrace{P; \ldots; P}_{\rho(x_i)\text{-mal}})$

Wir schreiben $\xrightarrow[LOOP]{}^*$ für 0 oder beliebig viele Berechnungsschritte.

Beispiel 10.1.3. *Wir betrachten das Programm*

$$\begin{array}{l} x_2 := x_1 + 1; \\ \textbf{LOOP } x_2 \textbf{ DO } x_3 := x_3 + 1 \textbf{ END} \end{array}$$

und die Variablenbelegung $\{x_1 \mapsto 2\}$. Dann gilt:

$$\begin{array}{ll} & (\{x_1 \mapsto 2\}, x_2 := x_1 + 1; \textbf{LOOP } x_2 \textbf{ DO } x_3 := x_3 + 1) \\ \xrightarrow[LOOP]{} & (\{x_1 \mapsto 2, x_2 \mapsto 3\}, \textbf{LOOP } x_2 \textbf{ DO } x_3 := x_3 + 1) \\ & \textit{da } (\{x_1 \mapsto 2\}, x_2 := x_1 + 1) \xrightarrow[LOOP]{} (\{x_1 \mapsto 2, x_2 \mapsto 3\}, \varepsilon) \\ \xrightarrow[LOOP]{} & (\{x_1 \mapsto 2, x_2 \mapsto 3\}, x_3 := x_3 + 1; x_3 := x_3 + 1; x_3 := x_3 + 1) \\ \xrightarrow[LOOP]{} & (\{x_1 \mapsto 2, x_2 \mapsto 3, x_3 \mapsto 1\}, x_3 := x_3 + 1; x_3 := x_3 + 1) \\ & \textit{da } (\{x_1 \mapsto 2, x_2 \mapsto 3\}, x_3 := x_3 + 1) \xrightarrow[LOOP]{} (\{x_1 \mapsto 2, x_2 \mapsto 3, x_3 \mapsto 1\}, \varepsilon) \\ \xrightarrow[LOOP]{} & (\{x_1 \mapsto 2, x_2 \mapsto 3, x_3 \mapsto 2\}, x_3 := x_3 + 1) \\ & \textit{da } (\{x_1 \mapsto 2, x_2 \mapsto 3, x_3 \mapsto 1\}, x_3 := x_3 + 1) \xrightarrow[LOOP]{} (\{x_1 \mapsto 2, x_2 \mapsto 3, x_3 \mapsto 2\}, \varepsilon) \\ \xrightarrow[LOOP]{} & (\{x_1 \mapsto 2, x_2 \mapsto 3, x_3 \mapsto 3\}, \varepsilon) \end{array}$$

10.1.3. LOOP-Berechenbarkeit

Definition 10.1.4 (LOOP-berechenbare Funktion)**.** *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt* LOOP-berechenbar*, wenn es ein LOOP-Programm P gibt, sodass für alle $n_1, \ldots, n_k \in \mathbb{N}$ und Variablenbelegungen $\rho = \{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$ gilt: $(\rho, P) \xrightarrow[LOOP]{}{}^* (\rho', \varepsilon)$ und $\rho'(x_0) = f(n_1, \ldots, n_k)$.*

Obige Definition zeigt, dass das LOOP-Programm die Eingaben dadurch empfängt, dass die Variablen $x_1, \ldots, x_k$ mit den entsprechenden Werten belegt sind. Es liefert die Ausgabe $f(n_1, \ldots, n_k)$ als Wert der Variablen x_0.

Beispiel 10.1.5. *Die Funktion $f(x_1) = x_1 + c$ ist LOOP-berechenbar. Das Programm $x_0 := x_1 + c$ belegt dies, denn $(\{x_1 \mapsto n_1\}, x_0 := x_1 + c) \xrightarrow[LOOP]{} (\{x_0 \mapsto n_1 + c, x_1 \mapsto n_1\}, \varepsilon)$ für alle $n_1 \in \mathbb{N}$.*

10.1.4. Eigenschaften von LOOP-Programmen und der LOOP-Berechenbarkeit

Satz 10.1.6. *Alle LOOP-Programme terminieren.*

Beweis. Dies gilt, da die Anzahl der $\xrightarrow[\text{LOOP}]{}$-Schritte begrenzt ist. Wir zeigen: Für alle (ρ, P) gibt es eine Zahl $j_{P,\rho} \in \mathbb{N}$ und ein ρ' mit $(\rho, P) \xrightarrow[\text{LOOP}]{}^{j_{P,\rho}} (\rho', \varepsilon)$.
Beweis mit Induktion über die Größe von P.

- Für $(\rho, x_i := x_j \pm c)$ wird genau 1 Schritt benötigt.
- Für Sequenzen $P_1; P_2$ und beliebige ρ, ρ' liefert die Induktionshypothese $j_{P_1,\rho}$ und $j_{P_2,\rho'}$ sowie ρ' und ρ'', sodass $(\rho, P_1; P_2) \xrightarrow[\text{LOOP}]{}^{j_{P_1,\rho}} (\rho', P_2) \xrightarrow[\text{LOOP}]{}^{j_{P_2,\rho'}} (\rho'', \varepsilon)$ und daher ist $j_{P,\rho} = j_{P_1,\rho} + j_{P_2,\rho'}$.
- Für **LOOP** x_i **DO** P **END** liefert die Induktionshypothese Zahlen j_{P,ρ_i} und Variablenumgebungen $\rho_2, \ldots, \rho_{n+1}$, sodass für jede Variablenumgebung ρ_1 gilt:

$$(\rho_1, \textbf{LOOP}\, x_i\, \textbf{DO}\, P\, \textbf{END}) \xrightarrow[\text{LOOP}]{} (\rho_1, P; \ldots; P) \xrightarrow[\text{LOOP}]{}^{j_{P,\rho_1}} (\rho_2, P; \ldots; P) \xrightarrow[\text{LOOP}]{}^{j_{P,\rho_2}} \ldots \xrightarrow[\text{LOOP}]{}^{i_{P,\rho_n}} (\rho_{n+1}, \varepsilon)$$

mit $n = \rho_1(x_i)$. Daher ist $j_{\textbf{LOOP}\, x_i\, \textbf{DO}\, P\, \textbf{END},\rho_1} = 1 + \sum_{k=1}^{\rho_1(x_i)} j_{P,\rho_i}$. □

Eine Konsequenz ist, dass alle LOOP-berechenbaren Funktionen *totale Funktionen* sind. Alleine daher ergibt sich, dass es turingberechenbare Funktionen gibt, die nicht LOOP-berechenbar sind. Z.B. die überall undefinierte Funktion. Wie wir sehen werden, gibt es intuitiv berechenbare Funktionen, die total sind, und trotzdem nicht LOOP-berechenbar sind (ein Beispiel ist die sogenannte Ackermannfunktion).

Wenn wir die Eingabevariablen kennen, dann können wir problemlos auch einen Befehl kodieren, der $x_i := c$ ausführt, indem wir $x_i := x_n + c$ verwenden, wobei x_n keine der Eingabevariablen ist und an keiner anderen Stelle im Programm verwendet wird. Der Begriff der Berechenbarkeit, sorgt dann dafür, dass x_n nicht in der initialen Variablenbelegung vorkommt und daher $\rho(x_n) = 0$ gilt. Ebenso können wir den Befehl $x_i := x_j$ kodieren, indem wir $x_i := x_j + 0$ verwenden. Daher verwenden wir beide Befehle als Kurzschreibweise. Ebenso verwenden wir

für Programm P die Schreibweise **IF** $x_i = 0$ **THEN** P **END** für das Programm

$$\begin{array}{l} x_n := 1; \\ \textbf{LOOP } x_i \textbf{ DO } x_n := 0 \textbf{ END}; \\ \textbf{LOOP } x_n \textbf{ DO } P \textbf{ END} \end{array}$$

wobei x_n eine Variable ist, die nicht in P und nicht in der Eingabe vorkommt. Beachte, dass

$$(\rho, x_n := 1; \textbf{LOOP } x_i \textbf{ DO } x_n := 0 \textbf{ END}; \textbf{LOOP } x_n \textbf{ DO } P \textbf{ END}) \xrightarrow[\text{LOOP}]{}{}^* (\rho\{x_n \mapsto 1\}, P)$$

falls $\rho(x_i) = 0$ und

$$\begin{array}{ll} & (\rho, x_n := 1; \textbf{LOOP } x_i \textbf{ DO } x_n := 0 \textbf{ END}; \textbf{LOOP } x_n \textbf{ DO } P \textbf{ END}) \\ \xrightarrow[\text{LOOP}]{}{}^* & (\rho\{x_n \mapsto 1\}, \underbrace{x_n := 0;}_{\rho(x_i) \text{ mal}} \textbf{LOOP } x_n \textbf{ DO } P \textbf{ END}) \\ \xrightarrow[\text{LOOP}]{}{}^* & (\rho\{x_n \mapsto 0\}, \textbf{LOOP } x_n \textbf{ DO } P \textbf{ END}) \\ \xrightarrow[\text{LOOP}]{}{}^* & (\rho\{x_n \mapsto 0\}, \varepsilon) \end{array}$$

falls $\rho(x_i) > 0$ gilt.

Analog schreiben wir **IF** $x_i = 0$ **THEN** P_1 **ELSE** P_2 **END** für

$$\begin{array}{l} x_n := 1; \\ x_m := 1; \\ \textbf{LOOP } x_i \textbf{ DO } x_n := 0 \textbf{ END}; \\ \textbf{LOOP } x_n \textbf{ DO } x_m := 0; P_1 \textbf{ END}; \\ \textbf{LOOP } x_m \textbf{ DO } P_2 \textbf{ END} \end{array}$$

wobei x_n, x_m Variablen sind, die nicht in der Eingabe vorkommen und auch sonst nicht im Programm verwendet werden.

Auch kompliziertere if-then-else-Bedingungen können entsprechend kodiert werden. Wir verwenden diese Abkürzungen daher, ohne sie explizit zu kodieren (und überlassen dies als Übungsaufgabe).

Die Operation zum Addieren zweier Variablen schreiben wir als $x_i := x_j + x_l$. Diese kann durch das folgende Programmstück simuliert werden:

$$\begin{array}{l} x_\ell := x_j; \\ \textbf{LOOP } x_k \textbf{ DO } x_\ell := x_\ell + 1 \textbf{ END}; \\ x_i := x_\ell \end{array}$$

wobei x_ℓ eine Variable ist, die nicht in der Eingabe vorkommt und auch sonst nicht im Programm verwendet wird. Dies zeigt auch, dass die Additionsfunktion $f(x_1, x_2) = x_1 + x_2$ LOOP-berechenbar ist.

Wird können andere Rechenoperationen (wie Multiplikation, mod (zur Berechnung des Rests einer ganzzahligen Division mit Rest), div (zur Berechnung des ganzzahligen Anteils einer ganzzahligen Division mit Rest), usw.) entsprechend definieren (wobei wir die Definitionen so anpassen, dass sie im Fall der Division durch 0 definiert sind und z.B. 0 als Ergebnis liefern) und verwenden diese im folgenden.

10.2. WHILE-Programme

10.2.1. Syntax der WHILE-Programme

WHILE-Programme erweitern LOOP-Programme um ein **WHILE**-Konstrukt, d.h. WHILE-Programme werden durch die kontextfreie Grammatik (V, Σ, P, Prg) erzeugt, wobei:

$$
\begin{array}{lcll}
V & = & \{Prg, Var, Id, Const\} & \\
\Sigma & = & \{\textbf{WHILE}, \textbf{LOOP}, \textbf{DO}, \textbf{END}, x, 0, \ldots, 9, ;, :=, +, -\} & \\
P & = & \{\ Prg & \to\ \textbf{WHILE}\ Var\ \neq 0\ \textbf{DO}\ Prg\ \textbf{END} \\
 & & & \mid \textbf{LOOP}\ Var\ \textbf{DO}\ Prg\ \textbf{END} \\
 & & & \mid Prg; Prg \\
 & & & \mid Var := Var + Const \\
 & & & \mid Var := Var - Const \\
 & & Var & \to\ x_{Id} \\
 & & Const & \to\ Id \\
 & & Id & \to\ 0 \mid 1 \mid \ldots \mid 9 \mid 1Id \mid 2Id \mid \ldots \mid 9Id\}
\end{array}
$$

10.2.2. Semantik der WHILE-Programme

Die Semantik von WHILE-Programmen wird definiert durch die schrittweise Ausführung mit Schritten $(\rho, P) \xrightarrow[\text{WHILE}]{} (\rho', P')$ wobei ρ, ρ' Variablenbelegungen sind, P ein WHILE-Programm ist und P' ein WHILE-Programm oder das leere Programm ε ist. Die meisten Regeln sind analog zur Semantik der LOOP-Programme. Im Wesentlichen neu sind zwei neue Regeln für die Auswertung des **WHILE**-Konstrukts.

Definition 10.2.1 (Berechnungsschritt $\xrightarrow[\text{WHILE}]{}$)**.** *Seien ρ, ρ' Variablenbelegungen und P, P WHILE-Programme oder das leere Programm ε. Ein Auswertungsschritt $\xrightarrow[WHILE]{}$ ist definiert durch die folgenden Fälle:*

- $(\rho, x_i := x_j + c) \xrightarrow[WHILE]{} (\rho\{x_i \mapsto n\}, \varepsilon)$ *wobei* $n = \rho(x_j) + c$
- $(\rho, x_i := x_j - c) \xrightarrow[WHILE]{} (\rho\{x_i \mapsto n\}, \varepsilon)$ *wobei* $n = \max(0, \rho(x_j) - c)$
- $(\rho, P_1; P_2) \xrightarrow[WHILE]{} (\rho', P_1'; P_2)$ *wenn* $(\rho, P_1) \xrightarrow[WHILE]{} (\rho', P_1')$ *und* $P_1' \neq \varepsilon$
- $(\rho, P_1; P_2) \xrightarrow[WHILE]{} (\rho', P_2)$ *wenn* $(\rho, P_1) \xrightarrow[WHILE]{} (\rho', \varepsilon)$
- $(\rho, \textbf{LOOP}\ x_i\ \textbf{DO}\ P\ \textbf{END}) \xrightarrow[WHILE]{} (\rho, \underbrace{P; \ldots; P}_{\rho(x_i)\text{-mal}})$
- $(\rho, \textbf{WHILE}\ x_i \neq 0\ \textbf{DO}\ P\ \textbf{END}) \xrightarrow[WHILE]{} (\rho, \varepsilon)$ *wenn* $\rho(x_i) = 0$
- $(\rho, \textbf{WHILE}\ x_i \neq 0\ \textbf{DO}\ P\ \textbf{END}) \xrightarrow[WHILE]{} (\rho, P; \textbf{WHILE}\ x_i \neq 0\ \textbf{DO}\ P\ \textbf{END})$ *wenn* $\rho(x_i) \neq 0$

Wir schreiben $\xrightarrow[\text{WHILE}]{}^*$ für 0 oder beliebig viele Berechnungsschritte.

10.2.3. WHILE-Berechenbarkeit

Definition 10.2.2 (WHILE-berechenbare Funktion)**.** *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt* WHILE-berechenbar*, wenn es ein WHILE-Programm P gibt, sodass*

- *für alle* $n_1, \ldots, n_k \in \mathbb{N}$*, sodass* $f(n_1, \ldots, n_k)$ *definiert ist: Für Variablenbelegung* $\rho = \{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$ *gilt* $(\rho, P) \xrightarrow[WHILE]{}^* (\rho', \varepsilon)$ *und* $\rho'(x_0) = f(n_1, \ldots, n_k)$*.*
- *Falls* $f(n_1, \ldots, n_k)$ *nicht definiert ist, so stoppt das Programm* P *nicht, d.h. für alle* $i \in \mathbb{N}$ *gibt es ein* ρ' *und ein* P'*, sodass* $(\rho, P) \xrightarrow[WHILE]{}^i (\rho', P')$*.*

Da jedes LOOP-Programm P auch ein WHILE-Programm ist, und für $(\rho, P) \xrightarrow[\text{LOOP}]{} (\rho', P')$ auch $(\rho, P) \xrightarrow[\text{WHILE}]{} (\rho', P')$ gilt, folgt:

Satz 10.2.3. *Jede LOOP-berechenbare Funktion ist auch WHILE-berechenbar.*

Wir hätten sogar das **LOOP**-Konstrukt aus der WHILE-Sprache herauslassen können, denn **LOOP** x_i **DO** P **END** kann durch das Programm

$$x_m := x_i;$$
$$\textbf{WHILE } x_m \neq 0 \textbf{ DO } x_m := x_m - 1; P \textbf{ END}$$

wobei x_m nirgends sonst vorkommt, simuliert werden.

Theorem 10.2.4. *Jede WHILE-berechenbare Funktion ist auch turingberechenbar.*

Beweis. In Abschnitt 9.3 haben wir gezeigt, dass die Konstrukte der WHILE-Programme (ohne **LOOP**-Konstrukt) durch Komposition von Mehrband-Turingmaschinen simuliert werden können. Dabei liegen die einzelnen Variablen x_i jeweils auf Band i der Mehrband-Turingmaschine. Die **LOOP**-Konstrukte können vorher mit der bereits vorgeschlagenen Konstruktion in **LOOP**-freie WHILE-Programme transformiert werden. □

10.3. GOTO-Programme

10.3.1. Syntax der GOTO-Programme

GOTO-Programme werden durch die CFG (V, Σ, P, Prg) erzeugt, wobei:

$$\begin{array}{lcll}
V & = & \{Prg, Var, Id, Const\} & \\
\Sigma & = & \{\textbf{GOTO}, \textbf{HALT}, \textbf{IF}, \textbf{THEN}, x, 0, \ldots, 9, :, ;, :=, +, -\} & \\
P & = & \{\ Prg & \to\ M_{Id}\text{: } \textbf{GOTO } M_{Id} \\
 & & & \mid M_{Id}\text{: } \textbf{IF } x_i = 0 \textbf{ THEN GOTO } M_{Id} \\
 & & & \mid M_{Id}\text{: } \textbf{HALT} \\
 & & & \mid M_{Id}\text{: } Var := Var + Const \\
 & & & \mid M_{Id}\text{: } Var := Var - Const \\
 & & & \mid Prg; Prg \\
 & & Var & \to\ x_{Id} \\
 & & Const & \to\ Id \\
 & & Id & \to\ 0 \mid 1 \mid \ldots \mid 9 \mid 1Id \mid 2Id \mid \ldots \mid 9Id\}
\end{array}$$

Die einzelnen Befehlszeilen von GOTO-Programmen sind mit Marken M_i: versehen. Wenn die Markierung unwichtig ist, lassen wir diese manchmal weg. Als Nebenbedingung (außerhalb der kontextfreien Grammatik) fordern wir, dass die einzelnen Programmzeilen mit

paarweise verschiedenen Markierungen markiert sind. Wir können Markierungen konsistent umbenennen, sodass wir davon ausgehen können, dass ein GOTO-Programm von der Form $M_1 : A_1; \ldots; M_n : A_n$ ist. Dies nennen wir ein *normalisiertes GOTO-Programm*.

10.3.2. Semantik der GOTO-Programme

Die Semantik von GOTO-Programmen wird definiert durch die schrittweise Ausführung mit Schritten $(\rho, P) \xrightarrow[\text{GOTO}]{P_0} (\rho', P')$ wobei ρ, ρ' Variablenbelegungen und P' ein GOTO-Programm oder das leere Programm ε ist, und P_0 das ursprüngliche GOTO-Programm ist.

Definition 10.3.1 (Berechnungsschritt $\xrightarrow[\text{GOTO}]{P_0}$)**.** *Sei P_0 ein normalisiertes GOTO-Programm, ρ eine Variablenbelegung und P ein GOTO-Programm. Dann ist ein Berechnungsschritt $\xrightarrow[\text{GOTO}]{P_0}$ durch die folgenden Fälle definiert:*

- $(\rho, M_i\colon x_j := x_k + c; P) \xrightarrow[\text{GOTO}]{P_0} (\rho', P)$, *wobei* $\rho' = \rho\{x_j \mapsto \rho(x_k) + c\}$
- $(\rho, M_i\colon x_j := x_k + c) \xrightarrow[\text{GOTO}]{P_0} (\rho', \varepsilon)$, *wobei* $\rho' = \rho\{x_j \mapsto \rho(x_k) + c\}$
- $(\rho, M_i\colon x_j := x_k - c; P) \xrightarrow[\text{GOTO}]{P_0} (\rho', P)$, *wobei* $\rho' = \rho\{x_j \mapsto \max(0, \rho(x_k) - c)\}$
- $(\rho, M_i\colon x_j := x_k - c) \xrightarrow[\text{GOTO}]{P_0} (\rho', \varepsilon)$, *wobei* $\rho' = \rho\{x_j \mapsto \max(0, \rho(x_k) - c)\}$
- $(\rho, M_i\colon \textbf{GOTO } M_k; P) \xrightarrow[\text{GOTO}]{M_1\colon A_1; \ldots; M_n\colon A_n} (\rho, M_k\colon A_k; \ldots; M_n\colon A_n)$ *wenn* $1 \leq k \leq n$
- $(\rho, M_i\colon \textbf{GOTO } M_k) \xrightarrow[\text{GOTO}]{M_1\colon A_1; \ldots; M_n\colon A_n} (\rho, M_k\colon A_k; \ldots; M_n\colon A_n)$ *wenn* $1 \leq k \leq n$
- $(\rho, M_i\colon \textbf{IF } x_r = 0 \textbf{ THEN GOTO } M_k; P) \xrightarrow[\text{GOTO}]{M_1\colon A_1; \ldots; M_n\colon A_n} (\rho, M_k\colon A_k; \ldots; M_n\colon A_n)$ *wenn* $1 \leq k \leq n$ *und* $\rho(x_r) = 0$
- $(\rho, M_i\colon \textbf{IF } x_r = 0 \textbf{ THEN GOTO } M_k) \xrightarrow[\text{GOTO}]{M_1\colon A_1; \ldots; M_n\colon A_n} (\rho, M_k\colon A_k; \ldots; M_n\colon A_n)$ *wenn* $1 \leq k \leq n$ *und* $\rho(x_r) = 0$
- $(\rho, M_i\colon \textbf{IF } x_r = 0 \textbf{ THEN GOTO } M_k; P) \xrightarrow[\text{GOTO}]{P_0} (\rho, P)$ *wenn* $\rho(x_r) \neq 0$
- $(\rho, M_i\colon \textbf{IF } x_r = 0 \textbf{ THEN GOTO } M_k) \xrightarrow[\text{GOTO}]{P_0} (\rho, \varepsilon)$ *wenn* $\rho(x_r) \neq 0$
- $(\rho, M_i\colon \textbf{HALT}; P) \xrightarrow[\text{GOTO}]{P_0} (\rho, M_i\colon \textbf{HALT})$
- $(\rho, \varepsilon) \xrightarrow[\text{GOTO}]{P_0} (\rho, \varepsilon)$

Beachte, dass der Fall $M_i\colon$ **HALT** nicht definiert ist, da dann kein weiterer Berechnungsschritt erfolgt. Der letzte Fall $(\rho, \varepsilon) \xrightarrow[\text{GOTO}]{P_0} (\rho, \varepsilon)$ ist dafür, dass ein Programm, welches nicht **HALT** ausführt, nicht anhält.

Wir schreiben $\xrightarrow[\text{GOTO}]{P_0}{}^*$ für 0 oder beliebig viele Berechnungsschritte und $\xrightarrow[\text{GOTO}]{P_0}{}^i$ für genau i Berechnungsschritte (mit dem Programm P_0).

10.3.3. GOTO-Berechenbarkeit

Definition 10.3.2 (GOTO-berechenbare Funktion)**.** *Eine Funktion* $f : \mathbb{N}^k \to \mathbb{N}$ *heißt* GOTO-berechenbar*, wenn es ein normalisiertes GOTO-Programm P gibt, sodass*

- *für alle* $n_1, \ldots, n_k \in \mathbb{N}$*, sodass* $f(n_1, \ldots, n_k)$ *definiert ist: Für Variablenbelegung* $\rho = \{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$ *gilt* $(\rho, P) \xrightarrow[GOTO]{P}{}^{*} (\rho', \mathbf{HALT})$ *und* $\rho'(x_0) = f(n_1, \ldots, n_k)$*.*
- *Falls* $f(n_1, \ldots, n_k)$ *nicht definiert ist, so stoppt das Programm P nicht, d.h. für alle* $i \in \mathbb{N}$ *gibt es* ρ' *und* P' *sodass* $(\rho, P) \xrightarrow[GOTO]{P}{}^{i} (\rho', P')$*.*

10.4. Äquivalenz der WHILE- und GOTO-Berechenbarkeit

Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden, denn

- Zuweisungen $x_j := x_k + c$ und $x_j := x_k - c$ und Sequenzen $P_1; P_2$ können auch als GOTO-Programm formuliert werden (durch zusätzliche Markierungen).
- **LOOP**-Schleifen können durch **WHILE**-Schleifen ersetzt werden
- jede **WHILE**-Schleife **WHILE** $x_i \neq 0$ **DO** P **END** kann durch

$$\begin{array}{ll} M_j\text{:} & \textbf{IF } x_i = 0 \textbf{ THEN GOTO } M_k \\ \ldots & P'; \\ M_{k-1}\text{:} & \textbf{GOTO } M_j; \\ M_k\text{:} & \ldots \end{array}$$

  simuliert werden, wobei P' das zu P zugehörige GOTO-Programm ist.

Sei P_W ein WHILE-Programm und P_G das oben beschriebene GOTO-Programm. Dann gilt: Falls $(\rho, P_W) \xrightarrow[\text{WHILE}]{}{}^{*} (\rho', \varepsilon)$, dann $(\rho, P_G) \xrightarrow[\text{GOTO}]{P_G}{}^{*} (\rho'', \mathbf{HALT})$ mit $\rho'(x_0) = \rho''(x_0)$, und umgekehrt. Dies kann per Induktion über die gegebene Folge von Berechnungsschritten bewiesen werden (wir lassen den genauen Beweis hier allerdings weg). Damit folgt, dass alle WHILE-berechenbaren Funktionen auch GOTO-berechenbar sind.

Sei $M_1\text{: } A_1; \ldots; M_n\text{: } A_n$ ein normalisiertes GOTO-Programm, sodass x_M nicht als Variable darin vorkommt. Das folgende WHILE-Programm simuliert das GOTO-Programm:

$$\begin{array}{l} x_M := 1; \\ \textbf{WHILE } x_M \neq 0 \textbf{ DO} \\ \quad \textbf{IF } x_M = 1 \textbf{ THEN } P_1 \textbf{ END}; \\ \quad \vdots \\ \quad \textbf{IF } x_M = n \textbf{ THEN } P_n \textbf{ END}; \\ \textbf{END} \end{array}$$

wobei P_i das folgende Programm ist:

- $x_j := x_k + c; x_M := x_M + 1$; falls $A_i = x_j := x_k + c$
- $x_j := x_k - c; x_M := x_M + 1$; falls $A_i = x_j := x_k - c$
- $x_M := j$; falls $A_i = \textbf{GOTO } M_j$
- **IF** $x_k = 0$ **THEN** $x_M := j$ **ELSE** $x_M := x_M + 1$ **END**; falls $A_i = \textbf{IF } x_k = 0 \textbf{ THEN GOTO } M_j$

- $x_M := 0$; falls $A_i =$ **HALT**

Dabei ist **IF-THEN-ELSE** wie vorher demonstriert, mittels **LOOP**-Konstrukten kodiert.

Sei $M_1\colon\ A_1; \ldots; M_n\colon\ A_n$ ein normalisiertes GOTO-Programm und P das oben beschriebene WHILE-Programm. Es lässt sich prüfen, dass aus $(\rho, M_1\colon\ A_1; \ldots; M_n\colon\ A_n) \xrightarrow[\text{GOTO}]{M_1\colon A_1;\ldots;M_n\colon A_n}{}^{*}$ $(\rho', \textbf{HALT})$ auch $(\rho, P) \xrightarrow[\text{WHILE}]{}{}^{*} (\rho''\{x_M \mapsto 0\}, \varepsilon)$ mit $\rho'(x_0) = \rho''(x_0)$ folgt, und umgekehrt. Damit folgt, dass jede GOTO-berechenbare Funktion auch WHILE-berechenbar ist.

Theorem 10.4.1. *WHILE- und GOTO-Berechenbarkeit sind äquivalente Begriffe. D.h. jede WHILE-berechenbare Funktion ist auch GOTO-berechenbar und umgekehrt ist jede GOTO-berechenbare Funktion auch WHILE-berechenbar.*

Satz 10.4.2. *WHILE-Programme benötigen nur eine* **WHILE***-Schleife.*

Beweis. Dies zeigen die Übersetzungen: Übersetze erst das WHILE-Programm in ein GOTO-Programm und anschließend übersetze das GOTO-Programm in ein WHILE-Programm. Dieses erzeugt nur eine **WHILE**-Schleife. □

10.5. Simulation von Turingmaschinen mit GOTO-Programmen

In diesem Abschnitt zeigen wir, dass GOTO-Programme Turingmaschinen simulieren können und können schließlich damit folgern, dass turingberechenbare Funktionen auch GOTO-berechenbar sind.

Die wesentliche Idee dabei ist, Konfigurationen der Turingmaschine mit Zahlen darzustellen, die in Programmvariablen abgelegt sind. Dazu betrachten wir zunächst, wie wir den nichtleeren Teil des Bandinhalts (der ein Wort über Γ^* ist) in eine Dezimalzahl konvertieren können.

10.5.1. Darstellung von Wörtern als natürliche Zahlen

Für ein Bandalphabet $\Gamma = \{a_1, \ldots, a_m\}$ identifizieren wir die Zeichen a_i mit ihrem Index i. Wörter $a_{i_1} \cdots a_{i_n}$ identifizieren wir daher mit der Folge $(i_1 \ldots i_n)$. Umgekehrt können wir aus Folgen $(i_1 \ldots i_n)$ wieder das Wort $a_{i_1} \cdots a_{i_n}$ rekonstruieren.

Sei $(i_1 \ldots i_n)$ eine solche Folge und sei $b > m$. Wir fassen $a_{i_1} \cdots a_{i_n}$ als Zahlen im Stellenwertsystem zur Basis b auf, wobei $a_1, \ldots, a_m$ Ziffern dieses Systems sind, wobei a_i die i-wertige Ziffer ist (damit kommt 0-wertige Ziffer nicht vor).

Dann kann der Wert von $a_{i_1} \cdots a_{i_n}$ berechnet werden als

$$(i_1 \cdots i_n)_b = \sum_{j=1}^{n} i_j \cdot b^{n-j}$$

Zur Berechnung von $(i_1 \cdots i_n)$ aus einer natürlichen Zahl (d.h. zur Berechnung der Umkehrfunktion von $(\cdots)_b$) teilt man die Dezimalzahl wiederholt mit Rest durch die Basis b bis kein Rest mehr entsteht. Man erhält daher Reste $r_0, \ldots, r_m$. Diese werden der Zahl $(r_m \cdots r_0)$ zur Basis b zugeordnet und repräsentieren daher das Wort $a_{r_m} \cdots a_{r_0}$.

Beispiel 10.5.1. *Sei* $\Gamma = \{\square, 0, 1\}$ *sodass* $a_1 = \square, a_2 = 0, a_3 = 1$. *Betrachte das Wort* $\square 110\square$, *d.h.* $a_1a_3a_3a_2a_1$. *Sei die Basis* $b = 4$. *Dann identifizieren wir* $\square 110\square$ *mit der Folge* (13321) *und* $(13321)_b = (13321)_4 = 1 \cdot 4^4 + 3 \cdot 4^3 + 3 \cdot 4^2 + 2 \cdot 4^1 + 1 \cdot 4^0 = 256 + 3 \cdot 64 + 3 \cdot 16 + 2 \cdot 4 + 1 \cdot 1 = 505$
Umgekehrt ergibt

$$\begin{array}{rcll} 505/4 & = & 126 & \textit{Rest } 1 \\ 126/4 & = & 31 & \textit{Rest } 2 \\ 31/4 & = & 7 & \textit{Rest } 3 \\ 7/4 & = & 1 & \textit{Rest } 3 \\ 1/4 & = & 0 & \textit{Rest } 1 \end{array}$$

und daher ergibt dies die Zahl (13321) *zur Basis* b, *welche das Wort* $a_1a_3a_3a_2a_1 = \square 110\square$ *repräsentiert.*

10.5.2. Darstellung von TM-Konfigurationen mit GOTO-Programmvariablen

Eine Konfiguration der TM über $\Gamma = \{a_1, \ldots, a_m\}$ schreiben wir als

$$a_{i_1} \cdots a_{i_n} z_k a_{j_1} \cdots a_{j_m}$$

und stellen diese im GOTO-Programm durch 3 Variablen x_p, x_z, x_s dar, wobei

$$\begin{array}{rcl} x_p & = & (i_1 \cdots i_n)_b \\ x_z & = & k \\ x_s & = & (j_m \cdots j_1)_b \end{array}$$

Beachte, dass wir für x_s die umgekehrte Reihenfolge der Ziffern verwenden.

Im Wesentlichen müssen wir für die TM-Simulation mit GOTO-Programmen nun Berechnungen implementieren, die für $\delta(z_k, a_{j_1}) = (z_l, a_{j'}, Q)$ mit $Q \in \{N, L, R\}$ entsprechende Aktualisierungen der Variablen x_p, x_z, x_s berechnen, sodass diese dem Übergangsschritt entsprechen.

Wir geben entsprechende GOTO-Programme an, die wir mit $DELTA(k, j_1)$ (für $\delta(z_k, a_{j_1})$) bezeichnen. Wir betrachten verschiedene Fälle.

- Falls $\delta(z_k, a_{j_1}) = (z_l, a_{j'}, L)$, dann ist der Übergang

  $$a_{i_1} \cdots a_{i_n} z_k a_{j_1} \cdots a_{j_m} \vdash a_{i_1} \cdots a_{i_{n-1}} z_l a_{i_n} a_{j'} a_{j_2} \cdots a_{j_m},$$

  falls $n > 0$. Wir betrachten den Spezialfall $n = 0$ später.

  Bezüglich der Zahlendarstellung gilt daher:
  - x_z muss auf den Wert l aktualisiert werden
  - x_p muss von $(i_1 \cdots i_n)_b$ zu $(i_1 \cdots i_{n-1})_b$ aktualisiert werden
  - x_s muss von $(j_m \cdots j_1)_b$ zu $(j_m \cdots j_2 j' i_n)_b$ aktualisiert werden

  Das Programm $DELTA(k, j_1)$ sei dann:

  $$\begin{array}{l} x_z := l; \\ x_s := x_s \text{ div } b; \\ x_s := b \cdot x_s + j'; \\ x_s := b \cdot x_s + (x_p \bmod b); \\ x_p := x_p \text{ div } b \end{array}$$

Eine Erläuterung dazu ist: Wir erhalten i_n durch Berechnung von x_p mod b. Abschneiden der letzten Stelle von x_p bewerkstelligen wir durch x_p div b. Für x_s schneiden wir erst die letzte Stelle ab (mit $x_s := x_s$ div b), und fügen j' als letzte Stelle hinzu (durch die Zuweisung $x_s := b \cdot x_s + j'$) und anschließend fügen wir i_n als letzte Stelle hinzu (durch die Zuweisung $x_s := b \cdot x_s + (x_p \bmod b)$).

Für den Grenzfall

$$z_k a_{j_1} \cdots a_{j_m} \vdash z_l \square a_{j_1} \cdots a_{j_m}$$

setze zuerst $x_p := r$; wobei $\square = a_r$ und führe dann das Programm für den allgemein Fall aus.

- Falls $\delta(z_k, a_{j_1}) = (z_l, a_{j'}, N)$, dann ist der Übergang

$$a_{i_1} \cdots a_{i_n} z_k a_{j_1} \cdots a_{j_m} \vdash a_{i_1} \cdots a_{i_n} z_l a_{j'} a_{j_2} \cdots a_{j_m}$$

und bezüglich der Zahlendarstellung gilt daher:

  - x_z muss auf den Wert l aktualisiert werden.
  - x_p bleibt unverändert.
  - x_s muss von $(j_m \cdots j_1)_b$ zu $(j_m \cdots j_2 j')_b$ aktualisiert werden.

Das Programm $DELTA(k, j_1)$ sei dann:

$$\begin{aligned} &x_z := l; \\ &x_s := x_s \text{ div } b; \\ &x_s := b \cdot x_s + j' \end{aligned}$$

- Falls $\delta(z_k, a_{j_1}) = (z_l, a_{j'}, R)$, dann ist der Übergang

$$a_{i_1} \cdots a_{i_n} z_k a_{j_1} \cdots a_{j_m} \vdash a_{i_1} \cdots a_{i_n} a_{j'} z_l a_{j_2} \cdots a_{j_m},$$

falls $m > 1$. Bezüglich der Zahlendarstellung gilt daher:

  - x_z muss auf den Wert l aktualisiert werden
  - x_p muss von $(i_1 \cdots i_n)_b$ zu $(i_1 \cdots i_n j')_b$ aktualisiert werden
  - x_s muss von $(j_m \cdots j_1)_b$ zu $(j_m \cdots j_2)_b$ aktualisiert werden

Das Programm $DELTA(k, j_1)$ sei dann:

$$\begin{aligned} &x_z := l; \\ &x_p := b \cdot x_p + j'; \\ &x_s := x_s \text{ div } b \end{aligned}$$

Für den Fall $m = 1$ gilt

$$a_{i_1} \cdots a_{i_n} z_k a_{j_1} \vdash a_{i_1} \cdots a_{i_n} a_{j'} z_l \square$$

In diesem Fall sei $DELTA(k, j_1)$ das Programm

$$\begin{aligned} &x_z := l; \\ &x_p := b \cdot x_p + j'; \\ &x_s := r \end{aligned}$$

wobei $\square = a_r$.

Beachte, dass wir die Sonderfälle „abfragen“ können, indem wir testen, ob x_p bzw. x_s den Wert 0 hat.

Insgesamt wird eine TM durch das folgende GOTO-Programm simuliert, welches seine Eingaben in den Programmvariablen $x_1, \ldots, x_k$ erwartet und seine Ausgabe in der Variablen x_0 ablegt: Das Programm gliedert sich in drei große Abschnitte:

$$M_1 : P_1;$$
$$M_2 : P_2;$$
$$M_3 : P_3$$

Programm P_1 erzeugt für die eingegebenen Variablenwerte in $x_1, \ldots, x_k$ deren Binärdarstellung und anschließend die Darstellung der TM-Konfiguration. Programm P_3 verwendet die Darstellung der Endkonfiguration, um die Ausgabe in der Ausgabevariablen x_0 zu erzeugen. Wir geben beide Programme nicht explizit an, da sie mit einfachen mod- und div-Berechnung durchgeführt werden können.

Programm P_2 führt schrittweise die Berechnung der Turingmaschine durch. Wir verwenden zur besseren Darstellung, Markierungen $M_{i,j}$, die durch Umbenennen wieder in die M_i-Form gebracht werden können:

M_2: $x_a := x_s \bmod b$;
IF $x_z = 1$ **and** $x_a = 1$ **THEN GOTO** $M_{1,1}$;
IF $x_z = 1$ **and** $x_a = 2$ **THEN GOTO** $M_{1,2}$;
$\vdots$
IF $x_z = q$ **and** $x_a = m$ **THEN GOTO** $M_{q,m}$;
$M_{1,1}$: $P_{1,1}$;
GOTO M_2;
$M_{1,2}$: $P_{1,2}$;
GOTO M_2;
$\vdots$
$M_{q,m}$: $P_{q,m}$
GOTO M_2

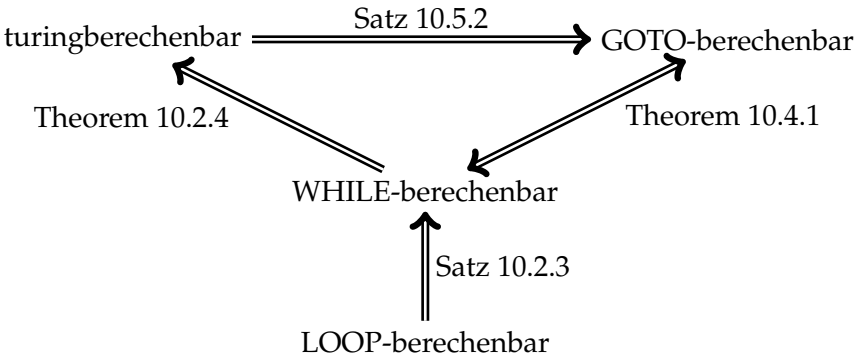
Die Programmabschnitte $P_{i,j}$ sind **GOTO** M_3, wenn z_i ein Endzustand und ansonsten $DELTA(i, j)$ (wie oben bereits festgelegt).

Insgesamt zeigt dieses GOTO-Programm:

Satz 10.5.2. *GOTO-Programme können Turingmaschinen simulieren. Jede turingberechenbare Funktion ist auch GOTO-berechenbar.*

Theorem 10.5.3. *Turingberechenbarkeit, WHILE-Berechenbarkeit und GOTO-Berechenbarkeit sind äquivalente Begriffe.*

Beweis. Das folgt aus den vorher gezeigten Sätzen und Theoremen. Das folgende Schaubild zeigt die gezeigten Implikationen, wobei wir der Vollständigkeit halber, auch die LOOP-Berechenbarkeit aufführen:


turingberechenbar
Satz 10.5.2
GOTO-berechenbar
Theorem 10.2.4
Theorem 10.4.1
WHILE-berechenbar
Satz 10.2.3
LOOP-berechenbar

□

11. ⋆ Primitiv und μ-rekursive Funktionen

In diesem Kapitel betrachten wir zwei weitere Formalismen, um Berechenbarkeit darzustellen. Anstelle von imperativen Kern-Programmiersprachen oder konkreten Maschinenmodelle betrachten wir mit rekursiven Funktionen einen eher mathematischen Begriff. Wir führen primitiv rekursive Funktionen und sogenannte μ-rekursive Funktionen ein und werden im Verlauf des Kapitels zeigen, dass die ersteren genau die LOOP-berechenbaren Funktionen darstellen, während die μ-rekursiven Funktionen äquivalent zu den turingberechenbaren Funktionen sind.

11.1. Primitiv rekursive Funktionen

Wir definieren die primitiv rekursiven Funktionen, um in Anschluss daran zu zeigen, dass diese genau durch die LOOP-Programme berechnet werden.

11.1.1. Definition der primitiv rekursiven Funktionen

Definition 11.1.1 (Primitiv rekursive Funktion)**.** *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist* primitiv rekursiv*, wenn sie der folgenden induktiven Definition genügt*[1]*:*

- *Jede konstante Funktion f mit $f(x_1, \ldots, x_k) = c \in \mathbb{N}$ ist primitiv rekursiv.*
- *Die Projektionsfunktionen π_i mit $\pi_i^k(x_1, \ldots, x_k) = x_i$ sind primitiv rekursiv.*
- *Für $k = 1$: Die Nachfolgerfunktion $succ(x) = x + 1$ ist primitiv rekursiv.*
- *Wenn $g : \mathbb{N}^m \to \mathbb{N}, h_1 : \mathbb{N}^k \to \mathbb{N}, \ldots, h_m : \mathbb{N}^k \to \mathbb{N}$ primitiv rekursiv sind, dann ist auch f mit $f(x_1, \ldots, x_k) = g(h_1(x_1, \ldots, x_k), \ldots, h_m(x_1, \ldots, x_k))$ primitiv rekursiv.*
- *Wenn $g : \mathbb{N}^{k-1} \to \mathbb{N}$ und $h : \mathbb{N}^{k+1} \to \mathbb{N}$ primitiv rekursiv sind, dann ist auch f mit*

$$f(x_1, \ldots, x_k) = \begin{cases} g(x_2, \ldots, x_k), & \text{wenn } x_1 = 0 \\ h(f(x_1 - 1, x_2, \ldots, x_k), x_1 - 1, x_2, \ldots, x_k), & \text{sonst} \end{cases}$$

primitiv rekursiv.

11.1.2. Einfache Beispiele und Konstruktionen primitiv rekursiver Funktionen

Die Projektionsfunktionen zusammen mit der Komposition erlauben die Konstruktion primitiv rekursiver Funktionen, welche Komponenten eines Tupels entfernen, vertauschen und verdoppeln. Z.B. ist für primitiv rekursives $g : \mathbb{N}^4 \to \mathbb{N}$ auch $f : \mathbb{N}^3 \to \mathbb{N}$ mit $f(n_1, n_2, n_3) = g(n_2, n_3, n_3, n_2)$ primitiv rekursiv, da f durch

$$f(n_1, n_2, n_3) = g(\pi_2^3(n_1, n_2, n_3), \pi_3^3(n_1, n_2, n_3), \pi_3^3(n_1, n_2, n_3), \pi_2^3(n_1, n_2, n_3))$$

[1] D.h. die Menge der primitiv rekursiven Funktionen ist die kleinste Menge von Funktionen, die alle genannten Eigenschaften erfüllt

dargestellt werden kann.
Für den rekursiven Fall muss die Rekursion nicht notwendigerweise durch das erste Argument laufen, da man mithilfe der Projektionen, der Komposition sowie zusätzlicher Hilfsfunktionen auch ein anderes Argument nehmen kann. Allgemein kann für $1 \leq i \leq k$

$$f(x_1,\ldots,x_k) = \begin{cases} g(x_1,\ldots,x_{i-1},x_{i+1},x_k), \text{falls } x_i = 0 \\ h(f(x_1,\ldots,x_{i-1},x_i-1,x_{i+1},\ldots,x_k),x_1,\ldots,x_{i-1},x_i-1,x_{i+1},\ldots,x_k), \text{sonst} \end{cases}$$

durch $f(x_1,\ldots,x_k) = f'(x_i,x_1\ldots,x_{i-1},x_{i+1},\ldots,x_k)$ dargestellt werden, wobei

$$\begin{array}{rcl} f'(x_1,\ldots,x_k) & = & \begin{cases} g(x_2,\ldots,x_k), \text{falls } x_1 = 0 \\ h'(f'(x_1-1,x_2\ldots,x_k),x_1-1,x_2,\ldots,x_k), \text{sonst} \end{cases} \\ h'(x_0,\ldots,x_k) & = & h(x_0,x_2,x_3,\ldots,x_i,x_1,x_{i+1},x_{i+2},\ldots,x_k) \end{array}$$

Daher verwenden wir die Rekursion auch für andere Argumente (d.h. nicht nur für das erste).

Beispiel 11.1.2. *Die Additionsfunktion $add(x_1,x_2) = x_1 + x_2$ ist primitiv rekursiv, da sie durch*

$$add(x_1,x_2) = \begin{cases} x_2, & \text{falls } x_1 = 0 \\ succ(add(x_1-1,x_2)), & \text{sonst} \end{cases}$$

definiert werden kann. Beachte, dass wir hierbei für die Definition der primitiven Rekursion für g und h die Funktionen $g = \pi^1_1$ und $h(x_1,x_2,x_3) = succ(\pi^3_1(x_1,x_2,x_3))$ verwendet haben. Wir geben im Folgenden die Funktionen g und h nicht immer explizit an.

Beispiel 11.1.3. *Für die Addition $x_1 + x_2$ haben wir x_1-mal die Nachfolgerfunktion auf x_2 angewendet. Für die Erstellung einer Multiplikationsfunktion $x_1 * x_2$ kann man analog verfahren und x_1-mal x_2 zu 0 addieren:*

$$mult(x_1,x_2) = \begin{cases} 0, & \text{falls } x_1 = 0 \\ add(mult(x_1-1,x_2),x_2), & \text{sonst} \end{cases}$$

Die Differenz $x_1 - x_2$ ist nicht primitiv rekursiv, da wir den undefinierten Fall $x_1 < x_2$ nicht behandeln können. Allerdings ist die angepasste Differenz (die $x_1 - x_2 = 0$ setzt, falls $x_1 < x_2$) primitiv rekursiv:

Beispiel 11.1.4. *Die Funktion $sub : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$ sei definiert durch*

$$sub(x_1,x_2) = \begin{cases} x_1, & \text{falls } x_2 = 0 \\ pred(sub(x_1,x_2-1)), & \text{sonst} \end{cases}$$

wobei

$$pred(x_1) = \begin{cases} 0, & \text{falls } x_1 = 0 \\ x_1 - 1, & \text{sonst} \end{cases}$$

Beachte, dass wir für sub die Rekursion im zweiten Argument angewendet haben. Da pred primitiv rekursiv ist, ist damit auch sub primitiv rekursiv.

11.1.3. Primitiv rekursive Funktionen zur Gödelisierung von Tupeln natürlicher Zahlen

In diesem Abschnitt beschäftigen wir uns damit, wie wir Tupel von natürlichen Zahlen $(x_1,\ldots,x_k)$ bijektiv in die natürlichen Zahlen (d.h. jedem Tupel wird bijektiv eine natürliche

Zahl zugeordnet) abbilden können. Das Ziel dabei ist es, *primitiv rekursive* Funktionen zu erstellen, die diese Abbildung und die Umkehrabbildung realisieren.

Im Rahmen des Abzählschemas für Paare natürlicher Zahlen haben wir bereits in Kapitel 2 Funktionen gesehen, die Paare natürlicher Zahlen in die natürlichen Zahlen abbilden.

Nun verwenden wir den Ansatz aus (Sch08) und verwenden die folgende Funktion, die Paare aus $\mathbb{N} \times \mathbb{N}$ bijektiv in die natürlichen Zahlen einbettet[2]:

$$c(x,y) = \binom{x+y+1}{2} + x$$

Z.B. zeigt die folgende Tabelle die Werte von $c(x,y)$ für $x,y \in \{0,\ldots,5\}$:

$x\backslash y$	0	1	2	3	4	5
0	0	1	3	6	10	15
1	2	4	7	11	16	22
2	5	8	12	17	23	30
3	9	13	18	24	31	39
4	14	19	25	32	40	49
5	20	26	33	41	50	60

Die Funktion c ist primitiv rekursiv, da $\binom{0}{2} = 0$ und $\binom{n+1}{2} = \binom{n}{2} + n$.

Um ein beliebiges $(k+1)$-Tupel $(x_0,\ldots,x_k)$ in eine natürliche Zahl injektiv abzubilden, können wir die Funktion c wiederholt anwenden. Wir definieren dies durch den Operator $\langle\cdot\rangle$, der definiert ist als: $\langle x_0,\ldots,x_k\rangle = c(x_0,c(x_1,\ldots,c(x_k,0)\ldots))$. Da die Komposition von primitiv rekursiven Funktionen wieder primitiv rekursiv ist, ist auch $\langle\cdot\rangle$ primitiv rekursiv.

Um aus einer mit $\langle\cdot\rangle$ erzeugten natürlichen Zahl wieder die ursprünglichen Zahlen des Tupels zu gewinnen, genügt es Funktionen *left* und *right* zu haben, die $\mathit{left}(c(x,y)) = x$ und $\mathit{right}(c(x,y)) = y$ berechnen: Dann können wir Funktionen d_i programmieren, sodass $d_i(\langle x_0,\ldots,x_k\rangle) = x_i$:

$$\begin{aligned} d_0(x) &= \mathit{left}(x) \\ d_1(x) &= \mathit{left}(\mathit{right}(x)) \\ d_i(x) &= \mathit{left}(\underbrace{\mathit{right}(\mathit{right}\ldots(\mathit{right}}_{i\text{-mal}}(x))\ldots)) \end{aligned}$$

Es bleibt zu zeigen, dass die Funktionen *left* und *right* existieren und primitiv rekursiv sind. Hierfür benötigen wir auch primitiv rekursive Prädikate. Wir nennen ein Prädikat primitiv rekursiv, wenn die ihm zugeordnete 0-1-wertige charakteristische Funktion primitiv rekursiv ist. Statt Prädikate direkt zu verwenden, benutzen wir im Folgenden direkt die 0-1-wertige charakteristische Funktion, d.h. k-stellige Prädikate P,Q seien Funktionen der Form $P,Q : \mathbb{N}^k \to \{0,1\}$.

[2]Zur Erinnerung: Der Binomialkoeffizient „n über k" für $n,k \in \mathbb{N}$ ist definiert als $\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!}$ falls $k \leq n$ und $\binom{n}{k} = 0$ falls $k > n$.

Der beschränkte Maximum-Operator $\overline{max}_P : \mathbb{N}^k \to \mathbb{N}$ für ein k-stelliges Prädikat $P : \mathbb{N}^k \to \{0,1\}$ sei definiert durch

$$\overline{max}_P(n, x_2, \ldots, x_k) := \begin{cases} \max\{x_1 \leq n \mid P(x_1, x_2, \ldots, x_k)\}, & \text{wenn es ein } 0 \leq x_1 \leq n \text{ gibt} \\ & \text{mit } P(x_1, x_2, \ldots, x_k) = 1 \\ 0, & \text{sonst} \end{cases}$$

D.h. der $\overline{max}_P(n, x_2, \ldots, x_k)$-Operator sucht nach einem maximalen Wert x_1, sodass $P(x_1, \ldots, x_k)$ erfüllt ist und x_1 nicht größer als n ist.

Wenn P primitiv rekursiv ist, dann ist auch $\overline{max}_P$ primitiv rekursiv:

$$\overline{max}_P(x_1, \ldots, x_k) = \begin{cases} 0, & \text{wenn } x_1 = 0 \\ x_1, & \text{wenn } P(x_1, \ldots, x_k) = 1 \\ \overline{max}_P(x_1 - 1, x_2, \ldots, x_k), & \text{sonst} \end{cases}$$

Äquivalent dazu ist die Definition:

$$\overline{max}_P(x_1, \ldots, x_k) = \begin{cases} 0, & \text{wenn } x_1 = 0 \\ \overline{max}_P(x_1 - 1, x_2, \ldots, x_k) & \\ \quad + P(x_1, \ldots, x_k) * (x_1 - \overline{max}_P(x_1 - 1, x_2, \ldots, x_k)), & \text{sonst} \end{cases}$$

Analog kann man einen beschränkten Existenztest $\overline{exists}_P : \mathbb{N}^k \to \{0,1\}$ programmieren, der für ein gegebenes Prädikat $P : \mathbb{N}^k \to \{0,1\}$ und eine Zahl n prüft, ob es ein x_1 mit $0 \leq x_1 \leq n$ gibt, sodass $P(x_1, x_2, \ldots, x_k)$ erfüllt ist:

$$\overline{exists}_P(x_1, \ldots, x_k) = \begin{cases} P(0, x_2, \ldots, x_k), & \text{wenn } x_1 = 0 \\ 1, & \text{wenn } P(x_1, \ldots, x_k) = 1 \\ \overline{exists}_P(x_1 - 1, x_2, \ldots, x_k), & \text{sonst} \end{cases}$$

bzw. äquivalent dazu ist:

$$\overline{exists}_P(x_1, \ldots, x_k) = \begin{cases} 0, & \text{wenn } x_1 = 0 \\ P(x_1, \ldots, x_k) + \overline{exists}_P(x_1 - 1, \ldots, x_k) & \\ \quad - P(x_1, \ldots, x_k) * \overline{exists}_P(x_1 - 1, \ldots, x_k), & \text{sonst} \end{cases}$$

Sei $eqc(m, x, y)$ das Prädikat, dass testet, ob $c(x, y) = m$ gilt. Das Prädikat eqc ist primitiv rekursiv: Berechne zunächst $c(x, y)$ und anschließend vergleiche m mit dem erhaltenen Wert (durch primitive Rekursion).

Nun können wir die Funktionen *left* und *right* definieren: Die Idee dabei ist, die folgende Implementierung zu verwenden:

$$\begin{aligned} left(n) &= \max\{x \leq n \mid \exists y \leq n : c(x, y) = n\} \\ right(n) &= \max\{y \leq n \mid \exists x \leq n : c(x, y) = n\} \end{aligned}$$

D.h. mit „Durchprobieren“ wird jeweils nach dem richtigem Wert für x bzw. y „gesucht“. Da die Abbildung $c(x, y)$ bijektiv ist, gibt es für festes n nur genau ein Wertepaar (x, y) sodass $c(x, y) = n$ gilt. Außerdem gilt, dass $x \leq c(x, y)$ und $y \leq c(x, y)$. Daher genügt es, die Werte bis

n für x und y zu testen.
Die Darstellung mit den zuvor definierten beschränkten Operatoren (und zusätzlichen Prädikaten P_1, P_2, Q_1, Q_2) ist:

$$\begin{array}{lcl} \mathit{left}(n) & = & \overline{\mathit{max}}_{P_1}(n,n) \\ P_1(x,n) & = & \overline{\mathit{exists}}_{P_2}(n,x,n) \\ P_2(y,x,n) & = & \mathit{eqc}(n,x,y) \\ \mathit{right}(n) & = & \overline{\mathit{max}}_{Q_1}(n,n) \\ Q_1(y,n) & = & \overline{\mathit{exists}}_{Q_2}(n,y,n) \\ Q_2(x,y,n) & = & \mathit{eqc}(n,x,y) \end{array}$$

Daher sind *left* und *right* primitiv rekursiv.

11.1.4. Äquivalenz von LOOP-berechenbaren und primitiv rekursiven Funktionen

Sei P ein LOOP-Programm. Seien $x_0, x_1, \ldots, x_n$ alle vom Programm P verwendeten Variablen (auch solche, die nicht in ρ vorkommen). Dann ist die von P berechnete Funktion vollständig durch eine Funktion $g_P(\langle x_0, \ldots, x_n\rangle) = \langle x_0', \ldots, x_n'\rangle$ beschrieben.
Wir zeigen zunächst, dass g_P primitiv rekursiv ist.

Lemma 11.1.5. *Für jedes LOOP-Programm P ist die zugehörige Funktion g_P primitiv rekursiv.*

Beweis. Wir zeigen, dass für jedes Teilprogramm Q die zugehörige Funktion g_Q primitiv rekursiv ist und verwenden Induktion über die Größe des Programms. Es gibt keine Programme der Größe 0, also müssen wir keine Basisfälle betrachten. Im Induktionsschritt gibt es folgende Möglichkeiten:

- Q ist eine Zuweisung $x_i = x_j \pm c$. Dann muss für die zugehörige Funktion g_Q gelten: $g_Q(\langle m_0, \ldots, m_n\rangle) = \langle m_0, \ldots, m_{i-1}, m_j \pm c, m_i, \ldots, m_n\rangle$. Diese Berechnung kann durch $g_Q(x) = \langle d_0(x), \ldots, d_{i-1}(x), d_j(x) + c, d_{i+1}(x), \ldots d_n(x)\rangle$ dargestellt werden und da Komposition, Addition, angepasste Subtraktion, $\langle\cdot\rangle$ und d_i primitiv rekursiv sind, ist auch g_Q primitiv rekursiv.
- Q ist eine Sequenz $Q_1; Q_2$. Dann liefert uns die Induktionshypothese primitiv rekursive Funktionen g_{Q_1} und g_{Q_2}. Die Funktion $g_Q(x) = g_{Q_2}(g_{Q_1}(x))$ ist primitiv rekursiv, da die Komposition primitiv rekursiver Funktionen wieder primitiv rekursiv ist.
- Q ist von der Form **LOOP** x_i **DO** P **END**. Dann liefert uns die Induktionshypothese eine primitiv rekursive Funktion g_P und wir konstruieren g_Q so, dass g_P dem Wert von x_i entsprechend oft angewendet wird:

$$\begin{array}{lcl} g_Q(x) & = & \mathit{run}(d_i(x), x) \\ \mathit{run}(n,x) & = & \begin{cases} x, & \text{falls } n = 0 \\ g_P(\mathit{run}(n-1,x)), & \text{sonst} \end{cases} \end{array}$$

  Die Funktion *run* ist primitiv rekursiv und damit ist ebenfalls g_Q primitiv rekursiv. □

Satz 11.1.6. *Jede LOOP-berechenbare Funktion ist primitiv rekursiv.*

Beweis. Sei $f : \mathbb{N}^k \to \mathbb{N}$ LOOP-berechenbar. Dann gibt es ein LOOP-Programm P, sodass für $\rho = \{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$ gilt $(\rho, P) \xrightarrow[\text{LOOP}]{}{}^* (\rho', \varepsilon)$ und $\rho'(x_0) = f(n_1, \ldots, n_k)$ und es

gilt $f(n_1, \ldots, n_k) = d_0(g_P(\langle 0, n_1, \ldots, n_k, 0, \ldots, 0\rangle))$. Da $\langle\cdot\rangle$, d_0 und g_P primitiv rekursiv sind, ist auch f primitiv rekursiv. □

Satz 11.1.7. *Jede primitiv rekursive Funktion ist LOOP-berechenbar.*

Beweis. Induktion über die Größe der primitiv rekursiven Funktion:

- Wenn f eine konstante Funktion, die Nachfolgerfunktion oder eine der Projektionsfunktionen ist, dann ist diese auch LOOP-berechenbar (es ist leicht ein entsprechendes LOOP-Programm anzugeben).
- Wenn f eine Komposition $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$ ist, dann liefert die Induktionshypothese LOOP-Programme P_h und $P_{g,1}, \ldots, P_{g,n}$, die h, g_1,…,g_n berechnen. Wir können das Programm für f nach dem Schema

  $$y_1 := g_1(x_1, \ldots, x_k); \ldots; y_n := g_n(x_1, \ldots, x_k); x_0 := h(y_1, \ldots, y_n)$$

  konstruieren. Für das genauere Vorgehen werden die Programme $P_{g,1}, \ldots, P_{g,n}, P_h$ so abgeändert, dass sie auf disjunkten Variablenmengen arbeiten und entsprechende Variableninhalte verdoppelt. (Wir lassen diese Details hier weg, überzeugen uns aber davon, dass ein entsprechendes Programm erstellt werden kann.)
- f ist rekursiv, d.h.

  $$f(x_1, \ldots, x_k) = \begin{cases} g(x_2, \ldots, x_k), & \text{wenn } x_1 = 0 \\ h(f(x_1 - 1, x_2, \ldots, x_k), x_1 - 1, x_2, \ldots, x_k), & \text{sonst} \end{cases}$$

  wobei $g : \mathbb{N}^{k-1} \to \mathbb{N}$ und $h : \mathbb{N}^{k+1} \to \mathbb{N}$ primitiv rekursiv sind. Dann erhalten wir durch die Induktionshypothese LOOP-Programme P_g, P_h, die g und h berechnet.

  Damit erstellen wir das Programm für f nach dem Schema

  $$y := 0; x_0 := g(x_2, \ldots, x_k); \textbf{LOOP } x_1 \textbf{ DO } x_0 := h(x_0, y, x_2, \ldots, x_k); y := y + 1 \textbf{ END}$$

□

Die Sätze 11.1.6 und 11.1.7 zeigen:

Theorem 11.1.8. *Die primitiv rekursiven Funktionen sind genau die LOOP-berechenbaren Funktionen.*

11.2. μ-Rekursive Funktionen

In diesem Abschnitt erweitern wir die Definition der rekursiven Funktionen um dem μ-Operator, was zu den μ-rekursiven Funktionen führt. Wir argumentieren im Anschluss, dass die μ-rekursiven Funktionen mit den WHILE-berechenbaren Funktionen (und damit u.a. auch mit den turingberechenbaren und den GOTO-berechenbaren) Funktionen übereinstimmen.

Der μ-Operator „sucht“ nach einer kleinsten Nullstelle einer gegebenen Funktion h. Wenn diese jedoch nicht existiert (entweder da h keine Nullstelle hat, oder da h undefiniert ist für Werte, die kleiner als die Nullstelle sind), dann ist auch der μ-Operator angewendet auf h undefiniert:

Definition 11.2.1 (μ-Operator)**.** *Sei $h : \mathbb{N}^{k+1} \to \mathbb{N}$ eine (partielle oder totale) Funktion. Dann ist $(\mu h) : \mathbb{N}^k \to \mathbb{N}$ definiert als*

$$(\mu h)(x_1, \ldots, x_k) = \begin{cases} n & \text{falls } h(n, x_1, \ldots, x_k) = 0 \text{ und für alle } m < n: \\ & h(m, x_1, \ldots, x_k) \text{ ist definiert und } h(m, x_1, \ldots, x_k) > 0. \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

Wie die folgende Definition festlegt, werden die μ-rekursiven Funktionen genauso wie die primitiv rekursiven Funktionen konstruiert, wobei zusätzlich der μ-Operator verwendet werden darf:

Definition 11.2.2 (μ-rekursive Funktion)**.** *Die Menge aller μ-rekursiven Funktionen sei die kleinste Menge, sodass gilt:*

- *Jede konstante Funktion f mit $f(x_1, \ldots, x_k) = c \in \mathbb{N}$ ist μ-rekursiv.*
- *Die Projektionsfunktionen π_i^k mit $\pi_i^k(x_1, \ldots, x_k) = x_i$ sind μ-rekursiv.*
- *Die Nachfolgerfunktion $succ(x) = x + 1$ ist μ-rekursiv.*
- *Wenn $g : \mathbb{N}^m \to \mathbb{N}, h_1 : \mathbb{N}^k \to \mathbb{N}, \ldots, h_m : \mathbb{N}^k \to \mathbb{N}$ μ-rekursiv sind, dann ist auch f mit $f(x_1, \ldots, x_k) = g(h_1(x_1, \ldots, x_k), \ldots, h_m(x_1, \ldots, x_k))$ μ-rekursiv.*
- *Wenn $g : \mathbb{N}^{k-1} \to \mathbb{N}$ und $h : \mathbb{N}^{k+1} \to \mathbb{N}$ μ-rekursiv, dann ist auch f mit*

  $$f(x_1, \ldots, x_k) = \begin{cases} g(x_2, \ldots, x_k) & \text{wenn } x_1 = 0 \\ h(f(x_1 - 1, x_2, \ldots, x_k), x_1 - 1, x_2, \ldots, x_k) & \text{sonst} \end{cases}$$

  μ-rekursiv.
- *Wenn $h : \mathbb{N}^{k+1} \to \mathbb{N}$ μ-rekursiv ist, dann ist auch $f = \mu h$ μ-rekursiv.*

Satz 11.2.3. *Jede WHILE-berechenbare Funktion ist μ-rekursiv.*

Beweis. Wir erweitern den Beweis von Satz 11.1.6, und betrachten nur den Fall einer **WHILE**-Schleife (alle anderen Fälle sind komplett analog zum Beweis von Satz 11.1.6).

- Falls P ein WHILE-Programm der Form **WHILE** $x_i \neq 0$ **DO** Q **END** ist, dann liefert die Induktionshypothese eine μ-rekursive Funktion g_Q für Q. Wir definieren:

  $$\begin{array}{lcl} g_P(x) & = & run(\mu(run_i)(x), x) \\ run_i(n, x) & = & d_i(run(n, x)) \\ run(n, x) & = & \begin{cases} x, & \text{falls } n = 0 \\ g_Q(run(n-1, x)) & \text{sonst} \end{cases} \end{array}$$

  Die Funktion $run(n, x)$ führt n-mal die zum Schleifenkörper zugehörige Funktion g_Q aus. Mit $\mu(run_i)(x)$ wird berechnet, wie oft die Schleife minimal durchlaufen werden muss, bis x_i den Wert 0 hat. Divergiert die Schleife, so ist $\mu(run_i)$ undefiniert und dementsprechend g_P undefiniert.

□

Satz 11.2.4. *Jede μ-rekursive Funktion ist WHILE-berechenbar.*

Beweis. Bis auf den μ-Operator ist der Beweis analog zum Beweis von Satz 11.1.7. Wir betrachten daher nur diesen neuen Fall.

- Sei f mit dem μ-Operator definiert, d.h. $f = \mu h$ für eine Funktion h. Dann liefert die Induktionshypothese ein WHILE-Programm P_h, welches h berechnet. Wir konstruieren P_f nach dem folgenden Schema:

  $x_0 := 0;$
  $y := h(0, x_1, \ldots, x_n);$
  WHILE $y \neq 0$ **DO**
  $\quad x_0 := x_0 + 1; y := h(x_0, \ldots, x_n)$
  END

  Hierbei werden die Berechnungen für $h(x_0, \ldots, x_n)$ durch das Programm P_h durchgeführt. Die **WHILE**-Schleife berechnet den minimalen Wert, sodass $h(x_0, \ldots, x_n) = 0$ gilt. Wenn dieser nicht existiert, so terminiert die Schleife nicht. Dies entspricht gerade der Berechnung von μh. □

Theorem 10.5.3 und Sätze 11.2.3 und 11.2.4 zeigen:

Theorem 11.2.5. *Die μ-rekursiven Funktionen entsprechen genau den WHILE-berechenbaren (und damit auch den GOTO- und turingberechenbaren) Funktionen.*

12. ⋆ Schnell wachsende Funktionen: Die Ackermannfunktion

In diesem Kapitel betrachten wir die von Wilhelm Ackermann in 1920er Jahren vorgeschlagene sogenannte Ackermannfunktion (Ack28), die extrem schnell wächst. Wir geben sie in einer von Rózsa Péter (Pét35) vereinfachten Variante an. Wir zeigen schließlich in diesem Kapitel, dass die Ackermannfunktion nicht LOOP-berechenbar (und damit auch nicht primitiv rekursiv) ist, aber WHILE-berechenbar ist. Da die Ackermannfunktion total ist, zeigt dies, dass die Klasse der totalen μ-rekursiven Funktionen echt größer ist als die Klasse der primitiv rekursiven Funktionen.

Die Ackermannfunktion $a : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$ sei definiert als

$$a(x,y) = \begin{cases} y+1, & \text{falls } x = 0 \\ a(x-1,1), & \text{falls } x > 0 \text{ und } y = 0 \\ a(x-1,a(x,y-1)), & \text{falls } x > 0 \text{ und } y > 0 \end{cases}$$

Einige Werte von $a(x,y)$ zeigen, wie schnell die Werte der Ackermannfunktion riesig groß werden:

$y \backslash x$	0	1	2	3	4
0	1	2	3	5	13
1	2	3	5	13	65533
2	3	4	7	29	$2^{65536} - 3$
3	4	5	9	61	$a(3, 2^{65536} - 3)$
4	5	6	11	125	$a(3, a(4,3))$

Die Ackermannfunktion ist total, denn die Rekursion terminiert stets: Entfaltet man die dritte Zeile jeweils für das zweite Argument, so erhält man

$$a(x,y) = \underbrace{(a(x-1,a(x-1,a(x-1,\ldots,a(x-1,}_{(y+1)\text{-mal}}1)\ldots))))$$

Da $a(x,y)$ für $x = 0$ terminiert und mit dem eben gezeigten alle rekursiven Aufrufe (nach einigen Schritten) das erste Argument echt verkleinern, folgt mit Induktion, dass $a(x,y)$ stets terminiert.

Intuitiv ist klar, dass man $a(x,y)$ berechnen kann, und dies auch mit einer üblichen modernen Programmiersprache tun kann. Wir zeigen, dass a WHILE-berechenbar ist.

Satz 12.0.1. *Die Ackermannfunktion a ist WHILE-berechenbar.*

Beweis. Wir erstellen ein WHILE-Programm, welches die rekursive Berechnung, durch eine Berechnung mit einer **WHILE**-Schleife und einem Stack bewerkstelligt. Der Stack ist eine Folge

von Zahlen $n_1, \ldots, n_k, 0$, wobei n_1 ganz oben liegt und die 0 am Ende den Stackboden darstellt. Wir stellen den Stack durch zwei Variablen dar: Die erste heißt *stacksize* und speichert die Stackgröße, die zweite namens *stack* speichert den Stackinhalt indem wir die Kodierungsfunktion $\langle\cdot\rangle$ auf ihn anwenden. Wir geben zunächst an, wie wir verschiedene Stack-Operationen als WHILE-Programm implementieren:

- $push(x, stack)$ legt die Zahl x oben auf den Stack. Als WHILE-Programm erstellen wir hierfür $stack := c(x, stack); stacksize := stacksize + 1;$.
- $x := pop(stack)$ entfernt das oberste Element vom Stack und setzt x auf dessen Wert. Als WHILE-Programm $x := left(stack); stack := right(stack); stacksize := stacksize - 1;$
- $stacksize \neq 1$ prüft ob der Stack nicht nur 1 Element enthält. Als WHILE-Programm können wir eine solche Abfrage formulieren.

Das Programm zur Berechnung des Wertes von $a(x, y)$ in der Variablen *result* hat dann folgende Form:

$stack := 0; stacksize := 0;$
$push(x, stack);$
$push(y, stack);$
WHILE $stacksize \neq 1$ **DO**
  $y := pop(stack);$
  $x := pop(stack);$
  IF $x = 0$
    THEN $push(y + 1, stack)$
    ELSE
      IF $y = 0$
        THEN $push(x - 1, stack); push(1, stack)$
        ELSE $push(x - 1, stack); push(x, stack); push(y - 1, stack)$
      END
  END
END;
$result := pop(stack)$

Zur Begründung der Korrektheit: Als Invariante gilt, dass der Stack $\langle n_1, \ldots, n_k\rangle$ die rekursiven Aufrufe $a(n_k, a(n_{k-1}, \ldots a(n_2, n_1)\ldots))$ darstellt. In der **WHILE**-Schleife werden die aktuellen Werte von y und x als oberste Elemente des Stacks gelesen. Je nachdem welcher Fall der Ackermannfunktion zutrifft wird fortgefahren: Wenn $x = 0$, dann wird $y + 1$ als Ergebnis auf den Stack gelegt. Wenn dies der letzte Aufruf von $a(x', y')$ war, enthält der Stack nur dieses Ergebnis und die **WHILE**-Schleife ist danach beendet. Anderenfalls steht das Ergebnis einer rekursiven Berechnung dadurch auf dem Stack zur Verfügung. Im Fall $y = 0$ werden neue Werte für x und y (d.h. $x - 1$ und 1) auf den Stack gelegt und der nächste Schleifendurchlauf startet damit die Berechnung von $a(x - 1, 1)$. Im allgemeinen Fall wird $y - 1, x, x - 1$ auf den Stack gelegt: Der nächste Schleifendurchlauf startet damit die Berechnung von $a(x, y - 1)$ und deren Ergebnis m liegt anschließend auf dem Stack. Dadurch startet im Anschluss die Berechnung von $a(m, x - 1)$. □

Wir zeigen nun, dass die Ackermannfunktion nicht LOOP-berechenbar ist. Zunächst beweisen wir einige Eigenschaften:

Lemma 12.0.2. *Die folgenden Monotonie-Eigenschaften gelten für die Ackermannfunktion a:*

1. $y < a(x,y)$
2. $a(x,y) < a(x,y+1)$
3. *Falls* $y \leq y'$*, dann gilt* $a(x,y) \leq a(x,y')$
4. *Falls* $y < y'$*, dann gilt* $a(x,y) < a(x,y')$
5. $a(x,y+1) \leq a(x+1,y)$
6. $a(x,y) < a(x+1,y)$
7. *Falls* $x \leq x'$ *und* $y \leq y'$*, dann gilt auch* $a(x,y) \leq a(x',y')$

Beweis. 1. Wir verwenden Induktion über das geordnete Paar (x,y) und die lexikographische Ordnung für Paare. Falls $x = (0,0)$, dann ist $0 < 0+1 = a(0,0)$. Für den Induktionsschritt sei $(x,y) > (0,0)$. Die Induktionshypothese ist, dass $y' < a(x',y')$ für alle $(x',y') < (x,y)$. Falls $y = 0$, dann gilt $a(x,0) = a(x-1,1)$ und mit der Induktionshypothese ist $1 < a(x-1,1)$. Daher gilt $0 < 1 < a(x,y)$. Falls $y > 0$, dann gilt $a(x,y) = a(x-1,a(x,y-1))$ und mit der Induktionshypothese angewendet für das Paar $(x-1,a(x,y-1))$ folgt $a(x,y-1) < a(x-1,a(x,y-1))$ und mit nochmaligen Anwenden für das Paar $(x,y-1)$ folgt $y-1 < a(x,y-1)$ und somit $y-1 < a(x,y-1) < a(x-1,a(x,y-1))$, was äquivalent zu $y \leq a(x,y-1) < a(x-1,a(x,y-1))$, ist. Damit folgt $y < a(x-1,a(x,y-1)) = a(x,y)$.

2. Wenn $x = 0$ ist, dann gilt $a(0,y) = y+1 < y+2 = a(0,y+1)$. Für $x > 0$ gilt $a(x,y+1) = a(x-1,a(x,y))$ aus der Definition der Ackermannfunktion. Mit Punkt 1 folgt daraus $a(x,y) < a(x,y+1)$.
3. Dieser Teil folgt fast direkt aus Punkt 2. Verwende Induktion über $y'-y$: Wenn $y'-y = 0$, dann gilt $y = y'$ und die Behauptung gilt. Falls $y'-y > 0$, dann betrachte $y'' = y'-1$. Die Induktionshypothese zeigt $a(x,y) \leq a(x,y'')$, da $y' > y$ gelten muss (und damit $y'' \leq y$). Punkt 2 zeigt $a(x,y'') \leq a(x,y')$, woraus $a(x,y) \leq a(x,y')$ folgt.
4. Dieser Teil ist analog zum Beweis von Punkt 3.
5. Wir verwenden Induktion über y. Wenn $y = 0$, dann gilt $a(x+1,0) = a(x,1)$ per Definition von a. Für $y > 0$ liefert die Induktionsannahme $a(x,y) < a(x+1,y-1)$. Aus Punkt 1 folgt $y < a(x,y)$ und damit auch $y+1 \leq a(x,y) < a(x+1,y-1)$. Aus Punkt 2 folgt $a(x+1,y-1) < a(x+1,y)$ und damit $y+1 \leq a(x+1,y)$. Schließlich folgt mit Punkt 3 und der Definition von a: $a(x,y+1) \leq a(x,a(x+1,y)) = a(x+1,y)$
6. Es gilt $a(x,y) < a(x,y+1)$ mit Punkt 2 und $a(x,y+1) \leq a(x+1,y)$ mit Punkt 5. Daher folgt $a(x,y) < a(x+1,y)$.
7. Analog zu Punkt 3 können wir aus Punkt 6 folgern, dass aus $x \leq x'$ auch $a(x,y) \leq a(x',y)$ folgt. Schließlich folgt mit Punkt 3 aus $y \leq y'$ auch $a(x',y) \leq a(x',y')$. □

Lemma 12.0.3. *Die beiden folgenden Eigenschaften gelten für die Ackermannfunktion:*

1. $a(1,y) = y+2$
2. $a(2,y) = 2y+3$

Beweis. 1. Induktion über y. Die Basis gilt, denn $a(1,0) = a(0,1) = 1+1 = 2$. Für den Induktionsschritt sei $y > 0$. Dann gilt $a(1,y) = a(0,a(1,y-1)) = 1+a(1,y-1) =_{I.H.} 1+(y-1)+2 = y+2$

2. Induktion über y. Für die Basis gilt $a(2,0) = a(1,1) = a(0,a(1,0)) = 1 + a(1,0) = 1 + a(0,1) = 1 + 2 = 3$. Für den Induktionsschritt sei $y > 0$. Es gilt $a(2,y) = a(1,a(2,y-1)) = (a(2,y-1)) + 2$ mit der Definition von a und mit Punkt 1. Mit der Induktionshypothese folgt $a(2,y-1)) + 2 = 2(y-1) + 3 + 2 = 2y + 3$. □

Sei P ein LOOP-Programm und seien $x_0, \ldots, x_k$ die im Programm vorkommenden Variablen. Wir ordnen P die folgende Funktion zu:

$$f_P(n) = \max \left\{ \sum_{i=0}^{k} \rho'(x_i) \;\middle|\; \sum_{i=0}^{k} \rho(x_i) \leq n, (\rho, P) \xrightarrow[\text{LOOP}]{}{}^{*} (\rho', \varepsilon) \right\}$$

D.h. wir bestimmen die maximale Zahl, die als Summe aller Endbelegungen ρ' der Variablen $x_0, \ldots, x_k$ zustande kommt, über alle initialen Variablenbelegung ρ, die in der Summe der Werte $\rho(x_0), \ldots, \rho(x_k)$ den Wert n nicht überschreiten.

Satz 12.0.4. *Für jedes LOOP-Programm P gibt es eine Konstante k, sodass $f_P(n) < a(k,n)$ für alle $n \in \mathbb{N}$.*

Beweis. Der Beweis erfolgt durch Induktion über die Größe von P.

- Wenn das Programm eine Zuweisung der Form $x_i := x_j + c$ ist und $c > 1$, dann transformiere das Programm in die Sequenz $x_i := x_j + 1; \underbrace{x_i := x_i + 1; \ldots x_i := x_i + 1}_{(c-1)\text{-mal}}$.

  Anschließend können wir für alle Zuweisungen der Form $x_i := x_j \pm c$ annehmen, dass sie von der Form $x_i := x_j + c$ mit $c \in \mathbb{Z}$ und $c \leq 1$ sind. Dann gilt für solche Zuweisungen $f_P(n) \leq 2n + 1$, da im maximalen Fall $\rho(x_k) = 0$ für $k \neq j$, $\rho(x_j) = n$ und $c = 1$ und damit $\rho'(x_i) = n + 1, \rho'(x_j) = n$ und $\rho'(x_k) = 0$ für $k \neq j$ und $k \neq i$ gilt. Mit Lemma 12.0.3 folgt $f_P(n) \leq 2n + 1 < 2n + 3 = a(2,n)$. D.h. die Aussage gilt mit $k = 2$.
- Wenn $P = P_1; P_2$, dann liefert die Induktionsannahme $f_{P_1}(n) < a(k_1,n)$ und $f_{P_2}(n) < a(k_2,n)$ für irgendwelche Konstanten k_1, k_2. Es gilt

$$\begin{aligned}
f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \\
&< a(k_2, f_{P_1}(n)) && \text{(mit Annahme)} \\
&< a(k_2, a(k_1, n)) && \text{(mit Lemma 12.0.2 Punkt 4 und Annahme)} \\
&\leq a(\max\{k_1,k_2\}, a(\max\{k_1,k_2\}+1, n)) && \text{(mit Lemma 12.0.2 Punkt 7)} \\
&= a(\max\{k_1,k_2\}+1, n+1) && \text{(Definition von } a\text{)} \\
&< a(\max\{k_1,k_2\}+2, n) && \text{(mit Lemma 12.0.2 Punkt 5)}
\end{aligned}$$

  Daher gilt die Behauptung für $k = \max\{k_1,k_2\} + 2$.
- Falls P eine LOOP-Schleife **LOOP** x_i **DO** Q **END** ist und x_i in Q vorkommt, dann ersetze **LOOP** x_i **DO** Q **END** durch $x_i' := x_i$; **LOOP** x_i' **DO** Q **END**; $x_i' := 0$, wobei x_i' eine neue Variable ist. Das ändert $f_P(n)$ nicht. Nehme daher an, dass x_i nicht in Q vorkommt. Dann liefert die Induktionsannahme k_Q mit $f_Q(n) < a(k_Q,n)$. Da $f_P(n)$ eine Maximumberechnung ist, sei ρ so gewählt, dass $f_P(n)$ maximal ist. Wenn $\rho(x_i) = 0$, dann ist $f_P(n) = n < a(0,n)$. Wenn $\rho(x_i) = 1$, dann ist $f_P(n) = f_Q(n) < a(k_Q,n)$. Sei $\rho(x_i) \geq 2$.

Da x_i nicht in Q vorkommt, gilt:

$$
\begin{aligned}
f_P(n) &\leq \underbrace{f_Q(\ldots(f_Q}_{\rho(x_i)\text{-mal}}(n-\rho(x_i))\ldots)+\rho(x_i) && \text{(da } x_i \text{ nicht vorkommt, kann } f_Q \text{ darauf nicht wirken)}\\
&\leq a(k_1, \underbrace{f_Q(\ldots(f_Q}_{(\rho(x_i)-1)\text{-mal}}(n-\rho(x_i))\ldots))+\rho(x_i)-1 && \text{(mit Lemma 12.0.2, Punkt 4 und die I.H. gilt } <\text{, daher } \leq \text{ und } -1)\\
&\vdots\\
&\leq a(k_1, a(k_1, \ldots, a(k_1, n-\rho(x_i))\ldots))\\
&< \underbrace{a(k_1, \ldots, a(k_1}_{(\rho(x_i)-1)\text{-mal}}, a(k_1+1, n-\rho(x_i))\ldots)) && \text{(mit Lemma 12.0.2, Punkt 4 und 6)}\\
&= a(k_1+1, n-\rho(x_i)+\rho(x_i)-1) && \text{(Definition von } a\text{)}\\
&= a(k_1+1, n-1)\\
&\leq a(k_1+1, n) && \text{(mit Lemma 12.0.2, Punkt 2)}
\end{aligned}
$$

□

Theorem 12.0.5. *Die Ackermannfunktion ist nicht LOOP-berechenbar (und damit auch nicht primitiv rekursiv).*

Beweis. Nimm an a ist LOOP-berechenbar. Dann ist auch $g(x) = a(x,x)$ LOOP-berechenbar und es gibt ein LOOP-Programm P, das $g(x_1)$ berechnet. Dann gilt $g(x) \leq f_P(x)$ (da $\rho'(x_0) = g(x)$ nach Ausführung von P gelten muss). Mit Satz 12.0.4 gibt es eine Konstante k, sodass $f_P(n) < a(k,n)$. Wenn P nun mit $\rho = \{x_1 \mapsto k\}$ gestartet wird, dann gilt $g(k) \leq f_P(k) < a(k,k) = g(k)$. Da $g(k) < g(k)$ ein Widerspruch ist, war unsere Annahme falsch und a ist nicht LOOP-berechenbar. □

Theorem 12.0.6. *Es gibt totale WHILE-berechenbare (bzw. GOTO-berechenbare, turingberechenbare, μ-rekursive) Funktionen, die nicht LOOP-berechenbar (bzw. primitiv rekursiv) sind.*

13. 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 13.0.1. *Eine Sprache $L \subseteq \Sigma^*$ heißt* entscheidbar, *wenn die charakteristische Funktion von L, $\chi_L : \Sigma^* \to \{0,1\}$ 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 $\chi'_L : \Sigma^* \to \{0,1\}$ mit*

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

berechenbar ist.

Für Mengen $L \subseteq \mathbb{N}$ 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 13.0.2. *Ein Sprache L ist genau dann entscheidbar, wenn L und $\overline{L}$ jeweils semi-entscheidbar sind.*

Beweis. Aus einer Turingmaschine, die χ_L berechnet, können leicht TMs konstruiert werden, die χ'_L und $\chi'_{\overline{L}}$ berechnen. Umgekehrt kann aus zwei TMs M_L und $M_{\overline{L}}$, die χ'_L und $\chi'_{\overline{L}}$ berechnen, eine TM konstruiert werden, die sich wie folgt verhält und damit χ_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 $M_{\overline{L}}$.
> Wenn diese akzeptiert, dann akzeptiere mit Ausgabe 0.
> Ansonsten erhöhe i um 1 und starte von neuem.

□

Korollar 13.0.3. *Wenn L entscheidbar ist, dann ist auch $\overline{L}$ entscheidbar.*

Definition 13.0.4. *Eine Sprache $L \subseteq \Sigma^*$ heißt* rekursiv aufzählbar, *falls $L = \emptyset$ oder falls es eine totale berechenbare Funktion $f : \mathbb{N} \to \Sigma^*$ gibt, sodass $L = \bigcup_{i \in \mathbb{N}} f(i)$. Man sagt dann „$f$ zählt L auf".*

Lemma 13.0.5. *Die Sprache Σ^* ist rekursiv aufzählbar.*

Beweis. Wir zeigen nur den Fall $|\Sigma| = 1$. Sei $\Sigma = \{a\}$ und $n \in \mathbb{N}$. Konstruiere eine 2-Band-TM, die auf einem Band die Eingabe n in Binärdarstellung erhält und auf dem anderen Band das leere Wort enthält. Anschließend zählt die TM die Binärzahl herunter und fügt bei jedem Herunterzählen ein a hinzu. Wenn der Binärzähler auf 0 ist, dann enthält das andere Band das Wort $f(n) = \underbrace{a \cdots a}_{n}$. □

Satz 13.0.6. *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 $i = 0, 1, 2, 3, \ldots$ **tue**
  wenn $f(i) = w$ **dann**
    stoppe und gib 1 aus

berechnet $\chi'_L(w)$.

Sei nun L eine semi-entscheidbare Sprache und M eine TM, die χ'_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 $\mathit{left}(n)$ und $\mathit{right}(n)$ 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 Σ^* (dieses wird zuvor von M' berechnet, siehe Lemma 13.0.5). 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:

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

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. □

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. $L(M) = L$).
- χ'_L ist Turing-, WHILE-, GOTO-berechenbar.
- Es gibt berechenbare Funktionen, die L als Wertebereich (nämlich die L aufzählende Funktion) bzw. als Definitionsbereich (nämlich χ'_L) haben.

Beachte, dass die rekursive Aufzählbarkeit im Allgemeinen nicht dasselbe ist wie Abzählbarkeit (siehe auch Abschnitt 2.4): Eine Sprache L ist abzählbar, wenn es eine totale Funktion $f : \mathbb{N} \to L$ gibt, sodass $\bigcup_{i\in\mathbb{N}} f(i) = L$. Der wesentliche Unterschied ist, dass Abzählbarkeit nicht die Berechenbarkeit von f fordert. Daher sind die Begriffe Abzählbarkeit und rekursive Aufzählbarkeit verschieden.

13.1. Gödelisierung von Turingmaschinen

Wir erläutern, wie Turingmaschinenbeschreibungen als Zahlen in Binärdarstellung, d.h. als Wörter 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 Γ und Z durchnummeriert sind:

- $\Gamma = \{a_0, \ldots, a_k\}$
- $Z = \{z_0, \ldots, z_n\}$

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 δ-zugehörigen Wörter, werden (in irgendeiner Reihenfolge) hintereinander geschrieben. Wir bezeichnen dieses Wort als w_δ.

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

$$M_w := \begin{cases} M, & \text{wenn } w = w_M \\ \widehat{M}, & \text{sonst} \end{cases}$$

13.2. Das Halteproblem

13.2.1. Unentscheidbarkeit des speziellen Halteproblems

Definition 13.2.1 (Spezielles Halteproblem)**.** *Das* spezielle Halteproblem *ist die Sprache*

$$K := \{w \in \{0,1\}^* \mid M_w \textit{ hält für Eingabe } w\}$$

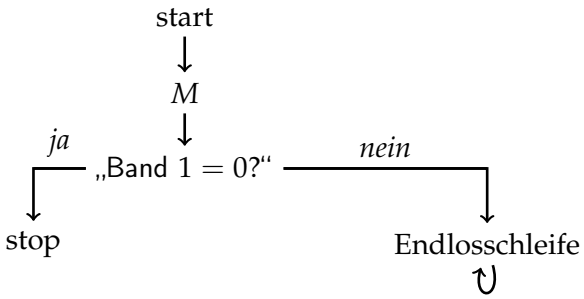
Satz 13.2.2. *Das spezielle Halteproblem ist nicht entscheidbar (und damit* unentscheidbar*).*

Beweis. Nimm an, K ist entscheidbar. Dann ist χ_K berechenbar. D.h. es gibt eine TM M, die χ_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. □

Der obige Beweis benutzt implizit ein Diagonalisierungsargument, das wir genauer erläutern: Betrachte die folgende Tabelle:

	w_1	w_2	w_3	w_4	$\dots$
M_{w_1}	*ja*	*nein*	*ja*	$\dots$	$\dots$
M_{w_2}	*nein*	*nein*	*ja*	$\dots$	$\dots$
M_{w_3}	*ja*	*nein*	*ja*	$\dots$	$\dots$
M_{w_4}	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$

Als Spaltenüberschriften enthält sie alle Wörter 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}	**ja**	*nein*	*ja*	...	...	...
M_{w_2}	*nein*	**nein**	*ja*	...	...	...
M_{w_3}	*ja*	*nein*	**ja**	...	...	...
M_{w_4}	...	...	...	...	...	...
...	...	...	...	...	...	...
M_{w_D}	*nein*	*ja*	*nein*	...	...	**?**
...						

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).

13.2.2. 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 13.2.3 (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 $f : \Sigma_1^* \to \Sigma_2^*$ gibt, sodass für alle $w \in \Sigma_1^*$ gilt: $w \in L_1 \iff f(w) \in L_2$.*

Satz 13.2.4. *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 χ_{L_2} berechenbar. Es gilt

$$\chi_{L_1}(w) = 1 \iff w \in L_1 \iff f(w) \in L_2 \iff \chi_{L_2}(f(w)) = 1$$

Damit ist $\chi_{L_1}(w) = \chi_{L_2}(f(w))$ berechenbar. □

Mit Kontraposition folgt aus Satz 13.2.4:

Lemma 13.2.5. *Sei $L_1 \leq L_2$ und L_1 ist unentscheidbar. Dann ist auch L_2 unentscheidbar.*

Wir verwenden meistens die Variante in Lemma 13.2.5, 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.

13.2.3. Unentscheidbarkeit des Halteproblems

Definition 13.2.6 ((Allgemeines) Halteproblem)**.** *Das (allgemeine)* Halteproblem *ist definiert als* $H := \{w\#x \mid TM\ M_w\ hält\ für\ Eingabe\ x\}$.

Satz 13.2.7. *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

$$\begin{array}{ll} & w \in K \\ \text{g.d.w.} & M_w \text{ hält für Eingabe } w \\ \text{g.d.w.} & w\#w \in H \\ \text{g.d.w.} & f(w) \in H \end{array}$$

Offensichtlich kann f durch eine TM berechnet werden und ist daher berechenbar. □

13.2.4. Halteproblem bei leerer Eingabe

Definition 13.2.8. *Das* Halteproblem auf leerem Band *ist die Sprache*

$$H_0 = \{w \mid M_w\ hält\ für\ die\ leere\ Eingabe\}$$

Satz 13.2.9. *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 $f(w_M\#x) = w_{M_x}$, wobei M_x die Turingmaschine ist, die sich wie M verhält aber vorher x auf das Eingabeband schreibt. Dann gilt

$$\begin{array}{ll} & w_M\#x \in H \\ \text{g.d.w.} & M \text{ hält für Eingabe } x \\ \text{g.d.w.} & M_x \text{ hält für die leere Eingabe} \\ \text{g.d.w.} & w_{M_x} \in H_0 \\ \text{g.d.w.} & f(w_M\#x) \in H_0 \end{array}$$

Die Funktion f kann durch eine TM berechnet werden und ist daher berechenbar. □

13.3. 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 13.3.1 (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*

$$C(\mathcal{S}) = \{w \mid die\ von\ M_w\ berechnete\ Funktion\ liegt\ in\ \mathcal{S}\}$$

unentscheidbar.

Beweis. Sei Ω 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.

$$\begin{aligned}
\text{Dann gilt: } w \in H_0 \quad &\Longrightarrow M_w \text{ hält auf leerer Eingabe} \\
&\Longrightarrow M_w^* \text{ berechnet } q \\
&\Longrightarrow \text{die von } M_w^* \text{ berechnete Funktion liegt in } \mathcal{S} \\
&\Longrightarrow f(w) \in C(\mathcal{S}) \\
\text{und ebenso: } w \notin H_0 \quad &\Longrightarrow M_w \text{ hält nicht auf leerer Eingabe} \\
&\Longrightarrow M_w^* \text{ berechnet } \Omega \\
&\Longrightarrow \text{die von } M_w^* \text{ berechnete Funktion liegt nicht in } \mathcal{S} \\
&\Longrightarrow f(w) \notin C(\mathcal{S})
\end{aligned}$$

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 13.0.3). □

13.4. Das Postsche Korrespondenzproblem

Wir betrachten in diesem Abschnitt ein weiteres unentscheidbares Problem, welches häufig verwendet wird, um dieses auf andere Probleme zu reduzieren, um deren Unentscheidbarkeit nachzuweisen. Das Problem ist das Postsche Korrespondenzproblem (PCP, Post Correspondence Problem) welches von Emil Post im Jahr 1946 vorgeschlagen wurde (Pos46).

Definition 13.4.1 (Postsches Korrespondenzproblem)**.** *Gegeben sei ein Alphabet Σ und eine Folge von Wortpaaren $K = ((x_1, y_1), \ldots, (x_k, y_k))$ mit $x_i, y_i \in \Sigma^+$. Das Postsche Korrespondenzproblem (PCP) ist die Frage, ob es für die gegebene Folge K eine nichtleere Folge von Indizes $i_1, \ldots, i_m$ mit $i_j \in \{1, \ldots, k\}$ gibt, sodass $x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$ gilt.*

Beachte, dass wir die zugehörige formale Sprache definieren können, indem wir genau die Sprache beschreiben, die alle richtigen Folgen von Indizes enthält. Dabei müssen die Folgen geeignet über einem Alphabet Σ dargestellt werden. Bezeichne $code(F) \in \Sigma^*$ diese Repräsentation von K. Dann ist

$$\begin{aligned}
\text{PCP} = \{code(K) \in \Sigma^* \mid &\text{für } K = ((x_1, y_1), \ldots, (x_k, y_k)) \text{ gibt es eine nichtleere Folge} \\
&\text{von Indizes } i_1, \ldots, i_m \text{ mit } i_j \in \{1, \ldots, k\}, \\
&\text{sodass } x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}\}
\end{aligned}$$

Das PCP kann als Spiel aufgefasst werden, wobei die einzelnen Spielsteinarten durch die Wortpaare dargestellt sind, d.h. die Spielsteine haben die Form $\left[\begin{array}{c} x_i \\ y_i \end{array}\right]$. 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 13.4.2. *Sei* $K = (\left[\begin{array}{c} a \\ aba \end{array}\right], \left[\begin{array}{c} baa \\ aa \end{array}\right], \left[\begin{array}{c} ab \\ bb \end{array}\right])$. *Dann ist* $I = 1, 2, 3, 2$ *eine Lösung, da*

$$\left[\begin{array}{c} a \\ aba \end{array}\right]\left[\begin{array}{c} baa \\ aa \end{array}\right]\left[\begin{array}{c} ab \\ bb \end{array}\right]\left[\begin{array}{c} baa \\ aa \end{array}\right] = abaaabbaa$$
$$= abaaabbaa$$

Für die Instanz $K = (\left[\begin{array}{c} ab \\ bba \end{array}\right], \left[\begin{array}{c} ba \\ baa \end{array}\right], \left[\begin{array}{c} ba \\ aba \end{array}\right], \left[\begin{array}{c} bba \\ b \end{array}\right])$ *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, 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.*

Wir zeigen die Unentscheidbarkeit des PCP durch zwei Reduktionen: Wir zeigen sowohl $H \leq$ MPCP als auch MPCP $\leq$ PCP. Dabei ist MPCP das *modifizierte* PCP, welches nur Lösungen erlaubt, die mit dem Spielstein der ersten Bauart beginnen.

13.4.1. MPCP $\leq$ PCP

Definition 13.4.3 (Modifiziertes Postsches Korrespondenzproblem)**.** *Gegeben sei ein Alphabet* Σ *und eine Folge von Wortpaaren* $K = ((x_1, y_1), \ldots, (x_k, y_k))$ *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* $1, i_2, \ldots, i_m$ *mit* $i_j \in \{1, \ldots, k\}$ *gibt, sodass* $x_1, x_{i_2} \cdots x_{i_m} = y_1, y_{i_2} \cdots y_{i_m}$ *gilt.*

Lemma 13.4.4. *MPCP* $\leq$ *PCP.*

Beweis. Wir müssen eine Funktion f angeben, sodass jede Instanz K des MPCP-Problems eine Lösung hat g.d.w. Instanz $f(K)$ eine Lösung für das PCP hat.
Sei $K = ((x_1, y_1), \ldots, (x_k, y_k))$ eine MPCP-Instanz über Alphabet Σ und seien \$ und # Symbole, die nicht in Σ 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 $f(K) = ((\bar{x}_1, \grave{y}_1), (\acute{x}_1, \grave{y}_1), \ldots (\acute{x}_k, \grave{y}_k), (\$, \#\$)) = (x'_1, y'_1), \ldots, (x'_{k+2}, y'_{k+2}))$.
Wenn $1, i_2, \ldots, i_m$ eine Lösung für das MPCP ist, dann sind die oberen und die unteren Wörter der Spielsteinfolge $(\bar{x}_1, \grave{y}_1)(\acute{x}_{i_2}, \grave{y}_{i_2}) \ldots, (\acute{x}_{i_m}, \grave{y}_{i_m})(\$, \#\$)$ ebenfalls gleich. D.h. $1, i_2 + 1, \ldots, i_m + 1, \ldots, k + 2$ ist eine Lösung für die PCP-Instanz $f(K)$. Umgekehrt gilt: Wenn $f(K)$ eine Lösung $(i_1, \ldots, i_m)$ hat, dann muss $i_1 = 1$ gelten, da sonst schon beim ersten Spielstein $\left[\begin{array}{c} x'_{i_1} \\ y'_{i_1} \end{array}\right]$ die beiden Wörter unterschiedlich anfangen. Ebenso muss $i_\ell = k + 2$ gelten für ein minimales ℓ. Alle Steine zwischendrin (i_j mit $2 \leq j < \ell$) müssen von der Form $(\acute{x}_u, \grave{y}_u)$ sein. Damit ist

klar, dass aus der Lösung für $f(K)$ auch eine MPCP-Lösung für K konstruiert werden kann: $i_1, i_2 - 1, \ldots, i_{\ell-1} - 1$. □

13.4.2. $H \leq$ MPCP

Lemma 13.4.5. *$H \leq MPCP$.*

Beweis. Sei $m\#w$ ein Wort bestehend aus einer Turingmaschinenbeschreibung m und einer Eingabe w. Wir müssen daraus eine Instanz für MPCP erstellen, sodass diese genau dann eine Lösung hat, wenn Turingmaschine M_m auf Eingabe w anhält. Der Beweis konstruiert ein MPCP, dass 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 $\left[\begin{array}{c} x_1 \\ y_1 \end{array}\right] = \left[\begin{array}{c} \# \\ \#z_0w\# \end{array}\right]$. Lösungsfolgen müssen daher mit diesem Wortpaar beginnen. Die weiteren Paare $\left[\begin{array}{c} x_2 \\ y_2 \end{array}\right], \ldots, \left[\begin{array}{c} x_k \\ y_k \end{array}\right]$ können durch vier Gruppen von Paaren beschrieben werden:

Kopierregeln:

- $\left[\begin{array}{c} a \\ a \end{array}\right]$ für alle $a \in \Gamma \cup \{\#\}$

Übergangsregeln:

- $\left[\begin{array}{c} za \\ z'c \end{array}\right]$ falls $\delta(z, a) = (z', c, N)$
- $\left[\begin{array}{c} za \\ cz' \end{array}\right]$ falls $\delta(z, a) = (z', c, R)$
- $\left[\begin{array}{c} bza \\ z'bc \end{array}\right]$ falls $\delta(z, a) = (z', c, L)$ für alle $b \in \Gamma$
- $\left[\begin{array}{c} \#za \\ \#z'\square c \end{array}\right]$ falls $\delta(z, a) = (z', c, L)$
- $\left[\begin{array}{c} z\# \\ z'c\# \end{array}\right]$ falls $\delta(z, \square) = (z', c, N)$
- $\left[\begin{array}{c} z\# \\ cz'\# \end{array}\right]$ falls $\delta(z, \square) = (z', c, R)$
- $\left[\begin{array}{c} bz\# \\ z'bc\# \end{array}\right]$ falls $\delta(z, \square) = (z', c, L)$ für alle $b \in \Gamma$

Löschregeln:

- $\left[\begin{array}{c} az_e \\ z_e \end{array}\right]$ für alle $a \in \Gamma, z_e \in E$
- $\left[\begin{array}{c} z_ea \\ z_e \end{array}\right]$ für alle $a \in \Gamma, z_e \in E$

Abschlussregeln:

- $\begin{bmatrix} z_e\#\# \\ \# \end{bmatrix}$ für alle $z_e \in E$

Wenn die Turingmaschine einen akzeptierenden Lauf hat, dann gibt es eine Folge von Konfigurationen $K_0 \vdash K_1 \vdash \ldots \vdash K_n$, 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 $\#K_0\#K_1\#\cdots\#K_n\#K_{n+1}\#\cdots\#K_m\#\#$ 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 $\#K_1\#K_2\#\cdots\#K_i\#$ und unten $\#K_1\#K_2\#\cdots\#K_i\#K_{i+1}\#$ 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) einen akzeptierenden Lauf ablesen, der bezeugt, dass die Turingmaschine bei Eingabe w hält.

Wegen der Kopienregeln können MPCP-Lösungen Wiederholungen von der Form $\#K\#K\#$ enthalten. Diese können vereinfacht werden, um einen Lauf zu erhalten.

Eine MPCP-Lösung kann auch mehreren Läufen hintereinander entsprechen. Dann betrachten wir nur den ersten Lauf.

Schließlich kann die MPCP-Instanz aus der Beschreibung $m\#w$ berechnet werden, d.h. es gibt eine berechenbar Funktion f mit $m\#w \in H \iff f(m\#w) \in \text{MPCP}$. Damit folgt $H \leq \text{MPCP}$. □

Satz 13.4.6. *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. □

13.4.3. Varianten und Bemerkungen zum PCP

Bereits für binäre Alphabete ist das PCP unentscheidbar:

Lemma 13.4.7 (01-PCP)**.** *Das Postsche Korrespondenzproblem über dem Alphabet Σ mit $|\Sigma| = 2$ (01-PCP) ist unentscheidbar.*

Beweis. Wir reduzieren PCP auf 01-PCP (und zeigen PCP $\leq$ 01-PCP). O.B.d.A. sei $\Sigma = \{0,1\}$. Sei $K = (x_1, y_1), \ldots, (x_k, y_k)$ eine Instanz des PCP über dem Alphabet $\{a_1, \ldots, a_j\}$. Sei $f(\varepsilon) = \varepsilon$, $f(a_i) = 10^i$, $f(a_i w) = f(a_i) f(w)$ und $f(K) = (f(x_1), f(y_1)), \ldots, (f(x_k), f(y_k))$. Dann ist $f(K)$

eine Instanz des 01-PCPs und offensichtlich gilt: $i_1, \ldots, i_n$ ist eine Lösung für K g.d.w. $i_1, \ldots, i_n$ ist eine Lösung für $f(K)$.

Die Funktion f ist offensichtlich turingberechenbar und daher folgt PCP $\leq$ 01-PCP und damit folgt aus der Unentscheidbarkeit von PCP, dass 01-PCP ebenfalls unentscheidbar ist. □

Lemma 13.4.8. *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 $\underbrace{i, \ldots, i}_{s\text{-mal}}, \underbrace{j, \ldots, j}_{r\text{-mal}}$, 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.

13.5. Universelle Turingmaschinen

Das PCP ist semi-entscheidbar, denn man kann einen Algorithmus angeben, der in einer inneren Schleife alle Folgen von i Spielsteinen aufzählt und in einer äußeren Schleife i wachsen lässt ($i = 1, 2, \ldots$). Der Algorithmus stoppt, wenn er eine Lösung gefunden hat. Der Algorithmus kann von einer Turingmaschine ausgeführt werden. Wenn eine Lösung der PCP-Instanz existiert, dann findet die Turingmaschine diese nach endlich vielen Schritten und akzeptiert. Existiert keine Lösung, so wird die Turingmaschine nicht anhalten.

Da wir bewiesen haben, dass $H \leq$ PCP gilt, folgt aus der Semi-Entscheidbarkeit von PCP auch die Semi-Entscheidbarkeit von H (siehe Satz 13.2.4). Das bedeutet, dass es eine Turingmaschine gibt, die sich bei Eingabe $w\#x$ so verhält wie M_w auf Eingabe x was das Halten betrifft. Ferner gibt es eine Turingmaschine U, die sich bei Eingabe $w\#x$ so verhält wie M_w auf Eingabe x, auch was die Ausgabe betrifft. Die Turingmaschine U nennt man eine *universelle Turingmaschine*, da sie sich wie ein Interpreter für Turingmaschinen verhält, d.h. sie wird durch die Eingabe w quasi programmiert und x ist dann die eigentliche Eingabe für das Programm.

13.6. ⋆ Unentscheidbarkeitsresultate für Grammatik-Probleme

Dieser Abschnitt wird nicht in der Vorlesung, aber eventuell in der Zentralübung behandelt. Er ist daher kein Prüfungsstoff.

In diesem Abschnitt beweisen wir die Unentscheidbarkeit von verschiedenen Problemen für Grammatiken. Als wesentliche Beweistechnik werden wir 01-PCP auf die verschiedenen Probleme reduzieren. Daher geben wir zunächst eine berechenbare Funktion F an, die eine Instanz von 01-PCP in zwei kontextfreie Grammatiken überführt:

Sei $K = ((x_1, y_1), \ldots, (x_k, y_k))$ über dem Alphabet $\Sigma = \{a, b\}$ (wir verwenden $\{a, b\}$ statt $\{0, 1\}$). Sei $F(K) = (\mathsf{G}_1, \mathsf{G}_2)$ wobei $\mathsf{G}_1, \mathsf{G}_2$ die folgenden kontextfreie Grammatiken sind:
Die Grammatik G_1 ist $\mathsf{G}_1 = (\{S, A, B\}, \Sigma \cup \{\$, 1, \ldots, k\}, P_1, S)$ wobei P_1 aus den folgenden Produktionen besteht:

$$\begin{aligned} S &\to A\$B \\ A &\to 1Ax_1 \mid \ldots kAx_k \\ A &\to 1x_1 \mid \ldots kx_k \\ B &\to \overline{y_1}B1 \mid \ldots \overline{y_k}Bk \\ B &\to \overline{y_1}1 \mid \ldots \overline{y_k}k \end{aligned}$$

Dabei ist $\overline{y_i}$ das Wort y_i in umgedrehter Form. $L(\mathsf{G}_1)$ enthält daher genau die Wörter

$$i_1 \cdots i_n x_{i_n} \cdots x_{i_1} \$ \overline{y_{j_1}} \cdots \overline{y_{j_m}} j_m \cdots j_1,$$

wobei $n, m \geq 1$ und $i_r, j_s \in \{1, \ldots, k\}$.
Die Grammatik G_2 ist $\mathsf{G}_2 = (\{S, T\}, \Sigma \cup \{\$, 1, \ldots, k\}, P_2, S)$, sodass P_2 aus den folgenden Produktionen besteht:

$$\begin{aligned} S &\to 1S1 \mid \ldots \mid kSk \mid T \\ T &\to a_1 T a_1 \mid \ldots\ a_n T a_n \mid \$ \end{aligned}$$

$L(\mathsf{G}_2)$ enthält daher genau die Wörter

$$i_1 \cdots i_n u \$ \overline{u} i_n \cdots i_1$$

mit $u \in \Sigma^*$, $i_j \in \{1, \ldots, k\}$, $n \geq 0$.
Wörter, die sowohl in $L(\mathsf{G}_1)$ als auch in $L(\mathsf{G}_2)$ liegen, sind daher von der Form

$$i_1 \cdots i_n x_{i_n} \cdots x_{i_1} \$ \overline{y_{i_1}} \cdots \overline{y_{i_n}} i_n \cdots i_1$$

mit $n \geq 1$ und $x_{i_1} \cdots x_{i_n} = y_{i_1} \cdots y_{i_n}$. Offensichtlich ist dann $i_1, \ldots, i_n$ eine Lösung von K. Umgekehrt gilt auch: Wenn $i_1, \ldots, i_n$ eine Lösung von K ist, dann ist das Wort $i_1 \cdots i_n x_{i_n} \cdots x_{i_1} \$ \overline{y_{i_1}} \cdots \overline{y_{i_n}} i_n \cdots i_1$ mit G_1 erzeugbar, und da dann $x_{i_1} \cdots x_{i_n} = y_{i_1} \cdots y_{i_n}$ gilt, ist es auch mit G_2 erzeugbar.
Insgesamt haben wir damit gezeigt:

Satz 13.6.1. *Das Schnittproblem für kontextfreie Grammatiken (also die Frage, ob $L(G_1) \cap L(G_2) = \emptyset$ für Grammatiken G_1, G_2 gilt), ist unentscheidbar.*

Beweis. Obige Funktion F zeigt, dass 01-PCP auf das Schnittproblem reduzierbar ist. Die Unentscheidbarkeit von 01-PCP impliziert daher die Unentscheidbarkeit des Schnittproblems. □

Wenn $L(\mathsf{G}_1) \cap L(\mathsf{G}_2) \neq \emptyset$, dann gilt auch $|L(\mathsf{G}_1) \cap L(\mathsf{G}_2)| = \infty$, da man die PCP-Lösung $i_1, \ldots, i_n$ Folge beliebig oft wiederholen kann, und entsprechende Wörter in den Grammatiken erzeugen kann. Daher gilt

Satz 13.6.2. *Das Problem, ob für kontextfreie Grammatiken G_1, G_2 gilt, $L(G_1) \cap L(G_2) = \infty$, ist unentscheidbar.*

Wenn $|L(\mathsf{G}_1) \cap L(\mathsf{G}_2)| = \infty$, dann ist $L(\mathsf{G}_1) \cap L(\mathsf{G}_2)$ eine nicht kontextfreie Sprache. Dies lässt sich mit dem Pumping-Lemma nachweisen.

Satz 13.6.3 (Unentscheidbarkeit des Schnittproblems für CFLs)**.** *Das Problem, ob der Schnitt zwei kontextfreier Sprachen kontextfrei ist, ist unentscheidbar.*

Beweis. Die Übersetzung F zeigt, dass ein 01-PCP K genau dann lösbar ist, wenn der Schnitt $L(\mathsf{G}_1) \cap L(\mathsf{G}_2)$ für $F(K) = (\mathsf{G}_1, \mathsf{G}_2)$ nicht kontextfrei ist. Daher haben wir das Komplement von 01-PCP auf das Problem aus dem Satz reduziert und aus der Unentscheidbarkeit von 01-PCP folgt die Unentscheidbarkeit des Komplements und damit folgt der Satz. □

Die Sprachen $L(\mathsf{G}_1)$ und $L(\mathsf{G}_2)$ sind nicht nur kontextfrei, sondern sogar deterministisch-kontextfrei (dies kann man zeigen, indem man entsprechende DPDAs konstruiert).

Da deterministisch-kontextfreie Sprachen unter Komplementbildung abgeschlossen sind und das Komplement auch berechnet werden kann (wir lassen den Beweis weg, er kann aber in der entsprechenden Literatur gefunden werden, und konstruiert aus einem DPDA für L einen DPDA für $\overline{L}$[1]), können wir deterministisch kontextfreie Grammatiken G'_1 und G'_2 berechnen mit $L(\mathsf{G}'_i) = \overline{\mathsf{G}_i}$. Sei F_i die Funktion, die G'_i aus G_i berechnet.

Satz 13.6.4. *Das Problem, ob für zwei gegebene kontextfreie Grammatiken $\mathsf{G}_A, \mathsf{G}_B$ gilt $L(\mathsf{G}_A) \subseteq L(\mathsf{G}_B)$ ist unentscheidbar.*

Beweis. Wir reduzieren das Komplement von 01-PCP auf das Inklusionsproblem. Sei F'' die Funktion, die für eine 01-PCP-Instanz K zunächst $F(K) = (\mathsf{G}_1, \mathsf{G}_2)$ und anschließend G_2 auf G'_2 abbildet (wobei $\mathsf{G}'_2 = F'(\mathsf{G}_2)$). Dann ist K unlösbar g.d.w. $L(\mathsf{G}_1) \subseteq L(\mathsf{G}'_2)$ gilt (da $L(\mathsf{G}_1) \cap L(\mathsf{G}_2) = \emptyset$ g.d.w. $L(\mathsf{G}_1) \subseteq \overline{L(\mathsf{G}_2)}$). Daher zeigt die Funktion F'', dass das Komplement von 01-PCP auf das Inklusionsproblem reduzierbar ist. Da das 01-PCP unentscheidbar ist, ist auch das Komplement von 01-PCP unentscheidbar, und daher ist auch Inklusionsproblem unentscheidbar. □

Satz 13.6.5 (Unentscheidbarkeit des Äquivalenzproblems für CFGs)**.** *Das Problem, ob für zwei gegebene kontextfreie Grammatiken $\mathsf{G}_A, \mathsf{G}_B$ gilt $L(\mathsf{G}_A) = L(\mathsf{G}_B)$ ist unentscheidbar.*

Beweis. Wir reduzieren das Komplement von 01-PCP auf das Äquivalenzproblems. Sei F'' die Funktion, die für eine 01-PCP-Instanz K zunächst $F(K) = (\mathsf{G}_1, \mathsf{G}_2)$ und anschließend G_1 auf G_3 und G_2 auf G'_2 abbildet, wobei G_3 die Vereinigung der Sprachen $L(\mathsf{G}_1)$ und $L(\mathsf{G}'_2)$ erzeugt (daher ist G_3 kontextfrei aber nicht notwendigerweise deterministisch kontextfrei). Dann gilt K ist unlösbar, g.d.w. $L(\mathsf{G}_3) = L(\mathsf{G}_2)'$ gilt (da $L(\mathsf{G}_3) = L(\mathsf{G}_1) \cup L(\mathsf{G}'_2)$). Daher zeigt die Funktion F'', dass 01-PCP auf das Äquivalenzproblem reduzierbar ist. Da das Komplement von 01-PCP unentscheidbar ist, ist auch Äquivalenzproblem unentscheidbar. □

Satz 13.6.6. *Das Problem, ob eine kontextfreie Grammatik G mehrdeutig ist, ist unentscheidbar.*

Beweis. Wir reduzieren 01-PCP auf das Mehrdeutigkeitsproblem. Sei $F_3(K)$ die Abbildung, die für jede 01-PCP-Instanz K zunächst $F(K) = (\mathsf{G}_1, \mathsf{G}_2)$ und anschließend G_4 mit $L(\mathsf{G}_4) = L(\mathsf{G}_1) \cup L(\mathsf{G}_4)$ berechnet. Dann gilt K ist lösbar, g.d.w. G_4 mehrdeutig ist. Denn nur wenn $L(\mathsf{G}_1) \cap L(\mathsf{G}_2) \neq \emptyset$ ist K lösbar und dann gibt es Wörter, die mit zwei unterschiedlichen Syntaxbäumen hergeleitet werden können (einmal mit G_1 und einmal mit G_2). □

[1] Die Schwierigkeit bei dieser Konstruktion liegt darin, dass der DPDA für L erst so modifiziert werden muss, dass er alle Wörter zu Ende liest und bei der Konstruktion des DPDAs für $\overline{L}$ muss sichergestellt werden, dass er in den Endzuständen keine ε-Übergänge macht.

Satz 13.6.7 (Unentscheidbarkeit des Regularitätsproblems für CFGs)**.** *Das Problem, ob für eine CFG G, $L(G)$ eine reguläre Sprache ist, ist unentscheidbar.*

Beweis. Sei F_4 die Abbildung, die für eine gegebene 01-PCP-Instanz K zuerst $F(K) = (\mathsf{G}_1, \mathsf{G}_2)$ und anschließend G_5 mit $L(\mathsf{G}_5) = \overline{L(\mathsf{G}_1)} \cup \overline{L(\mathsf{G}_2)}$ berechnet (dies ist möglich, da $\mathsf{G}_1, \mathsf{G}_2$ deterministisch kontextfrei sind).

F_4 zeigt, dass das Komplement von 01-PCP auf das Regularitätsproblem reduzierbar ist: Wenn K unlösbar (also K im Komplement von 01-PCP) dann ist $L(\mathsf{G}_5) = \Sigma^*$ (da $L(\mathsf{G}_1) \cap L(\mathsf{G}_2) = \emptyset$ in diesem Fall) und daher ist $L(\mathsf{G}_5)$ regulär. Wenn K lösbar ist, dann gilt wie bereits gezeigt $L(\mathsf{G}_1) \cap L(\mathsf{G}_2)$ ist nicht kontextfrei (und damit auch nicht regulär). Da $\overline{L(\mathsf{G}_5)} = \overline{\overline{L(\mathsf{G}_1)} \cup \overline{L(\mathsf{G}_2)}} = L(\mathsf{G}_1) \cap L(\mathsf{G}_2)$ ist $\overline{L(\mathsf{G}_5)}$ auch nicht regulär. Da reguläre Sprachen unter Komplementbildung abgeschlossen sind, kann $L(\mathsf{G}_5)$ auch nicht regulär sein.

Da 01-PCP unentscheidbar, ist auch das Komplement von 01-PCP unentscheidbar und damit folgt, dass das Regularitätsproblem unentscheidbar ist. □

Teil III.

Komplexitätstheorie

14. Einleitung, 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.

14.1. 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 „Verwirf"-Zustände erlauben, die zwar nicht akzeptierend sind, aber auch (genau wie Konfigurationen mit akzeptierenden Zuständen) keine Nachfolgekonfiguration 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 Einbandmaschinen 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 Mehr-

bandmaschine 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 14.1.1. *Sei M eine stets anhaltende Mehrband-DTM mit Startzustand z_0. Für Eingabe w definieren wir*

$time_M(w) := i$, *wenn für die Startkonfiguration für z_0 und w nach i Schritten ein Endzustand oder Verwirf-Zustand erreichbar ist*

Aufbauend auf diesem Begriff der Rechenzeit, definieren wir nun die Klasse von Sprachen $TIME(f(n))$:

Definition 14.1.2. *Für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ sei die Klasse $TIME(f(n))$ genau die Menge der Sprachen L, 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 gröberen Begriff, indem sie nur fordert, dass f ein beliebiges Polynom ist. Wir definieren zunächst, was ein Polynom ist:

Definition 14.1.3. *Ein* Polynom *ist eine Funktion $p : \mathbb{N} \to \mathbb{N}$ der Form:*

$$p(n) = \sum_{i=0}^{k} a_i \cdot n^i = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 \cdot n + a_0$$

mit $a_i \in \mathbb{N}$ und $k \in \mathbb{N}$. (Man spricht auch von einem Polynom des Grades *k.)*

Definition 14.1.4 (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.

Sei M eine DTM mit $time_M(w) \leq f(|w|)$. Unsere Äquivalenzbeweise zeigen, dass die Turingmaschinen durch GOTO-Programme simuliert werden können (wobei endlich viele Schritte im GOTO-Programm notwendig sind, um einen Turingmaschinenschritt zu simulieren). Das GOTO-Programm benötigt daher $O(f(|w|))$ Schritte. Die GOTO-Programme können in ein WHILE-Programm transformiert werden mit einer WHILE-Schleife, wobei die Anzahl der Durchläufe durch die WHILE-Schleife nicht größer sein kann als $f(|w|)$. Daher kann die WHILE-Schleife sogar durch das LOOP-Konstrukt $y := f(|w|)$; **LOOP** y **DO** ... **END** ersetzt werden. Das bedeutet: Wenn f selbst LOOP-berechenbar ist, sind auch Funktionen mit der Laufzeitkomplexität $f(n)$ noch LOOP-berechenbar. Daher sind z.B. Komplexitätsklassen $TIME(2^n)$ oder sogar $TIME(\underbrace{2^{2^{2^{\cdots^2}}}}_{n\text{-mal}})$ noch in der Klasse der LOOP-berechenbaren Funktionen enthalten.

14.1.1. Uniformes vs. logarithmisches Kostenmaß

Im allgemeinen macht es keinen asymptotischen Unterschied, ob wir eine Turingmaschine, oder die WHILE-Sprache oder die GOTO-Sprache verwenden, um ein Problem als effizient lösbar zu zeigen. Allerdings müssen wir dabei etwas aufpassen, wenn wir mit großen Zahlen umgehen. Eine Zuweisung der Form $x_i := x_j$ haben wir bisher in der operationalen Semantik der WHILE- oder GOTO-Sprachen mit einem Schritt gezählt, unabhängig davon, wie groß die in der Variablenumgebung stehenden Zahlen sind. Geht man so vor, so spricht man vom uniformen Kostenmaß. Solange alle jemals verwendeten Zahlen eine feste Konstante nicht überschreiten, so kann man dieses Vorgehen verwenden. Anderenfalls muss man zum *logarithmischen Kostenmaß* wechseln: Dieses verbucht für eine solche Zuweisung $x_i := x_j$ die Anzahl an notwendigen Bits zur Darstellung von $\rho(x_j)$ als Rechenschritte (dies passt zur Turingmaschine: diese muss so viele Bits kopieren). Beachte, dass eine Dezimalzahl d, mit $\lceil \log_2(d) \rceil$ Bits dargestellt werden kann.

Betrachte z.B. das WHILE-Programm

$x_0 := 2;$
WHILE $x_1 \neq 0$ **DO**
  $x_0 := x_0 * x_0;$
  $x_1 := x_1 - 1;$
END

Dieses berechnet $2^{2^{x_1}}$ als Ergebnis in der Ausgabevariablen x_0 (da die Schleife den Wert von x_0 genau x_1-mal quadriert). Mit uniformen Kostenmaß, läuft die Algorithmus in Zeit $O(x_1)$. Die dazu äquivalente Turingmaschine, kann das Ergebnis $2^{2^{x_1}}$ jedoch keinesfalls in weniger als 2^{x_1} Schritten berechnen, da das Ergebnis schon so viele Bits benötigt und diese auf das Band geschrieben werden müssen.

Wendet man jedoch das logarithmische Kostenmaß an, so wird dies wieder ausgeglichen (z.B. verbraucht dann der letzte Schleifendurchlauf mindestens 2^{x_1} Schritte, da die Zuweisung $x_0 := x_0 * x_0$ entsprechend viele Bits benötigt.

Falls notwendig, verwenden wir das logarithmische Kostenmaß, wenn die verwendeten Zahlen jedoch durch eine Konstante beschränkt sind, verwenden wir das uniforme Kostenmaß auch für Algorithmen in der WHILE-Sprache.

14.2. 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 anhalten. 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 14.2.1. *Sei M eine stets anhaltende Mehrband-NTM, die für jede Eingabe anhält. Dann definieren wir die Laufzeit[1] von M als*

$$ntime_M(w) := \max\{i \mid \text{für die Startkonfiguration für } z_0 \text{ und } w \text{ hält } M \text{ nach } i \text{ Schritten}\}$$

Schließlich definieren wir basierend auf diesem Laufzeitmaß die Komplexitätsklasse $NTIME(f(n))$ und die Klasse $\mathcal{NP}$:

Definition 14.2.2. *Für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ bezeichne $NTIME(f(n))$ die Klasse aller Sprachen L, für die es eine nichtdeterministische, stets anhaltende 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

$$\mathcal{NP} = \bigcup_{p \text{ Polynom}} NTIME(p(n))$$

14.3. Das P-vs.-NP-Problem

Lemma 14.3.1. *Es gilt $TIME(f(n)) \subseteq NTIME(f(n))$ 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 $ntime_M(w)$ und es gilt für diese Turingmaschine $time_M(w) = ntime_M(w)$. Damit folgt, dass aus $L \in TIME(f(n))$ auch $L \in NTIME(f(n))$ folgt. Schließlich zeigt dies auch $\mathcal{P} \subseteq \mathcal{NP}$. □

[1] Es gibt hierbei verschiedene Möglichkeiten dieser Definition, welche verschiedene Turingmaschinen erlauben oder verbieten. Diese Varianten führen jedoch zum selben Begriff der Klasse $\mathcal{NP}$.

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 Traveling-Salesperson-Problem) leicht ist zu zeigen, dass sie in $\mathcal{NP}$ liegen, aber noch niemand einen Algorithmus gefunden hat, der die Probleme in deterministischer Polynomialzeit löst, 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 Kapitel 15 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. 14.1 gezeigte Mengendiagramm helfen. Im gezeigten Diagramm sind alle Teilmengenbeziehungen als echt bekannt bis auf die unterste zwischen $\mathcal{NP}$ und $\mathcal{P}$.

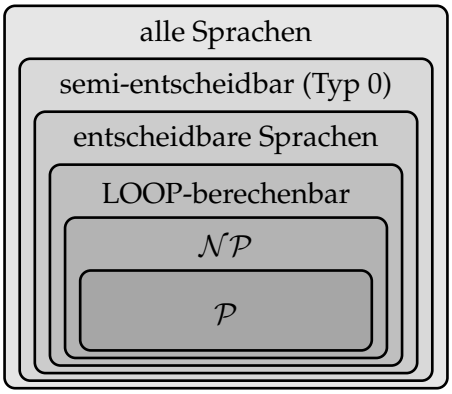


Abbildung 14.1.: Lage der Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$

15. NP-Vollständigkeit

15.1. Definition der NP-Vollständigkeit

15.1.1. Polynomialzeit-Reduktionen

In Abschnitt 13.2.2 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 f, sodass $w \in L_1$ g.d.w. $f(w) \in L_2$ für alle Wörter $w \in \Sigma_1^*$. 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 $time_M(w) \in O(|w|^k)$ für ein festes k.

Definition 15.1.1 (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 $L_1 \leq_\mathrm{p} L_2$), falls es eine totale und in deterministischer Polynomialzeit berechenbare Funktion $f : \Sigma_1^* \to \Sigma_2^*$ 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 13.2.2 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 $L_1 \leq_\mathrm{p} L_2$ 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 15.1.2. *Falls $L_1 \leq_\mathrm{p} L_2$ und $L_2 \in \mathcal{P}$, dann gilt $L_1 \in \mathcal{P}$. Ebenso gilt: Falls $L_1 \leq_\mathrm{p} L_2$ und $L_2 \in \mathcal{NP}$, dann gilt $L_1 \in \mathcal{NP}$.*

Beweis. Sei $L_1 \leq_\mathrm{p} L_2$ 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 polynomielle 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 erste 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 $M_f; M_2'$. □

Lemma 15.1.3. *Die Relation $\leq_\mathrm{p}$ ist transitiv, d.h. wenn $L_1 \leq_\mathrm{p} L_2$ und $L_2 \leq_\mathrm{p} L_3$, dann gilt auch $L_1 \leq_\mathrm{p} L_3$.*

Beweis. Der Beweis ist analog zum vorherigen und benutzt im Wesentlichen die Eigenschaft, dass die Komposition von zwei Polynomen immer noch ein Polynom ist. □

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 15.1.4 ($\mathcal{NP}$-Vollständigkeit)**.** *Eine Sprache L heißt* $\mathcal{NP}$-vollständig*, wenn gilt*

1. *$L \in \mathcal{NP}$ und*
2. *L ist* $\mathcal{NP}$-schwer[1]*: Für alle $L' \in \mathcal{NP}$ gilt $L' \leq_\mathrm{p} L$.*

Die Zugehörigkeit zu $\mathcal{NP}$ für eine betrachtete Sprache L nachzuweisen ist im Allgemein nicht so schwer. 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 $\leq_\mathrm{p}$ ausnutzen können und zeigen $L_0 \leq_\mathrm{p} L_1$. Da L_0 als $\mathcal{NP}$-vollständig bekannt ist, gilt für alle $L' \in \mathcal{NP}$: $L' \leq_\mathrm{p} L_0$ und aufgrund der Transitivität von $\leq_\mathrm{p}$ damit auch $L' \leq_\mathrm{p} L_1$ für alle $L' \in \mathcal{NP}$.

Bevor wir dieses eine Problem direkt als $\mathcal{NP}$-vollständig beweisen (als Satz von Cook im nächsten Abschnitt), zeigen wir den folgenden Satz:

Satz 15.1.5. *Sei L ein $\mathcal{NP}$-vollständiges Problem. Dann gilt $L \in \mathcal{P} \iff \mathcal{P} = \mathcal{NP}$.*

Beweis. Die „$\Longleftarrow$“-Richtung ist trivial. Für die andere Richtung: 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 $L' \leq_\mathrm{p} L$ und Lemma 15.1.2 folgt $L' \in \mathcal{P}$. Da dies für alle $L' \in \mathcal{NP}$ gilt, folgt $\mathcal{P} = \mathcal{NP}$. □

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.

[1] manchmal auch $\mathcal{NP}$-hart genannt

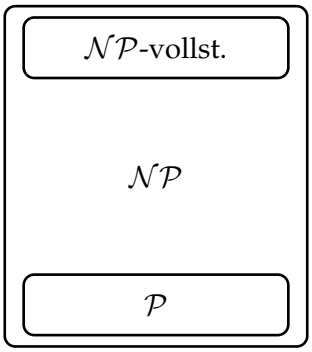


Abbildung 15.1.: Vermutete Lage der Probleme in $\mathcal{P}$, $\mathcal{NP}$ und der $\mathcal{NP}$-vollständigen Probleme

In Abb. 15.1 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 $\mathcal{NP}$-vollständig bekannt ist.

15.2. Satz von Cook

In diesem Abschnitt weisen wir nach, dass das sogenannte *Erfüllbarkeitsproblem der Aussagenlogik* (kurz SAT) $\mathcal{NP}$-vollständig ist. Nachdem wir die $\mathcal{NP}$-Vollständigkeit dieses Problems nachgewiesen haben, können wir es verwenden, um die $\mathcal{NP}$-Schwere anderer Probleme mithilfe von Polynomialzeit-Reduktionen einfacher nachzuweisen.

Wir definieren zunächst das SAT-Problem in Form eines gegeben/gefragt-Problems:

Definition 15.2.1 (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 Σ dargestellt werden. Bezeichne $\mathit{code}(F) \in \Sigma^*$ diese Repräsentation von F. Dann ist

$$\text{SAT} = \{\mathit{code}(F) \in \Sigma^* \mid F \text{ ist eine erfüllbare Formel der Aussagenlogik}\}$$

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 15.2.2. *$SAT \in \mathcal{NP}$.*

Beweis. Sei $\mathit{code}(F)$ 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 $\{x_1, \ldots, x_n\}$. Anschließend verwendet die Maschine den Nichtdeterminismus, um eine Belegung $I : \{x_1, \ldots, x_n\} \to \{0, 1\}$ 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:

$$M \text{ akzeptiert eine Formel } F \text{ g.d.w. } \mathit{code}(F) \in \text{SAT}.$$

Jeder Berechnungspfad der Turingmaschine läuft in Polynomialzeit in der Größe der Eingabe (d.h. in $|\mathit{code}(F)|$), 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. □

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.

Bevor wir den zweiten Teil der $\mathcal{NP}$-Vollständigkeit von SAT betrachten, beweisen wir einen Hilfssatz:

Lemma 15.2.3. *Für aussagenlogische Variablen $\{x_1, \ldots, x_n\}$ gibt es eine aussagenlogische Formel $\mathit{exactlyOne}(x_1, \ldots, x_n)$, sodass $I(\mathit{exactlyOne}(x_1, \ldots, x_n)) = 1$ g.d.w. I genau eine der Variablen x_i auf 1 setzt und alle anderen auf 0.*

Dabei ist die Größe der Formel $\mathit{exactlyOne}(x_1, \ldots, x_n)$ in $O(n^2)$.

Beweis. Die Formel

$$\mathit{exactlyOne}(x_1, \ldots, x_n) := (x_1 \vee \ldots \vee x_n) \wedge \bigwedge_{1 \le i < j \le n} \neg(x_i \wedge x_j)$$

leistet das verlangte: Die Disjunktion $x_1 \vee \ldots \vee x_n$ sichert zu, dass mindestens eine der Variablen $x_1, \ldots, x_n$ 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 $x_1, \ldots, x_n$ wahr sein darf). Die Größe der Formel ist $O(n^2)$. □

Wir betrachten nun den zweiten Teil, um den Nachweis der $\mathcal{NP}$-Vollständigkeit von SAT zu vervollständigen – die $\mathcal{NP}$-Schwere von SAT. Dieser Beweis ist schwierig, da wir zeigen müssen: Wenn wir SAT lösen können, dann können wir *jedes* andere Problem L aus $\mathcal{NP}$ auch lösen, indem wir es in Polynomialzeit auf SAT reduzieren. Das bedeutet: Wir müssen für jedes x-beliebige Problem L aus $\mathcal{NP}$ nachweisen, dass $L \le_{\text{p}} \text{SAT}$ gilt.

Da L ein Problem aus $\mathcal{NP}$ 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 15.2.4. *SAT ist $\mathcal{NP}$-schwer.*

Beweis. Sei $L \in \mathcal{NP}$ 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 $\Gamma = \{b_1, \ldots, b_l\}, Z = \{z_0, \ldots, z_k\}$ und z_0 der Startzustand. Die Formel F enthält die folgenden aussagenlogischen Variablen, die wir zunächst erläutern:

- $State_{t,z}$ für $t = 0, 1, \ldots, p(n)$ und $z \in Z$. Dabei soll gelten: $State_{t,z} = 1$ g.d.w. nach t Schritten ist TM M im Zustand z.
- $Pos_{t,i}$ für $t = 0, 1, \ldots, p(n)$,$i = -p(n), \ldots, p(n)$. 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 $t = 0, 1, \ldots, p(n)$, $i = -p(n), \ldots, p(n)$, $b = b_1, \ldots, b_l$. 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 $-p(n), \ldots, p(n)$ des Bands zu betrachten.

Die Formel F lässt sich in vier Unterformeln aufteilen:

$$F = \mathit{Rand} \wedge \mathit{Anfang} \wedge \mathit{Übergang} \wedge \mathit{Ende}$$

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 $\bigwedge_{t \in \{0,\ldots,p(n)\}} \mathit{exactlyOne}(State_{t,z_0}, \ldots, State_{t,z_k})$ zu.
  - Zu jedem Zeitpunkt t befindet sich der Kopf von M in genau einer Position auf dem Band. Das sichert die Formel $\bigwedge_{t \in \{0,\ldots,p(n)\}} \mathit{exactlyOne}(Pos_{t,-p(n)}, \ldots, Pos_{t,p(n)})$ zu.
  - Zu jedem Zeitpunkt t befindet sich in jeder Bandzelle genau ein Symbol aus Γ. Das sichert die Formel $\bigwedge_{t \in \{0,\ldots,p(n)\}} \bigwedge_{i \in \{-p(n),\ldots,p(n)\}} \mathit{exactlyOne}(Tape_{t,i,b_1}, \ldots, Tape_{t,i,b_l})$ zu.

  Daher setzen wir:

$$\mathit{Rand} := \bigwedge_{t \in \{0,\ldots,p(n)\}} \begin{pmatrix} \mathit{exactlyOne}(State_{t,z_0}, \ldots, State_{t,z_k}) \\ \wedge \mathit{exactlyOne}(Pos_{t,-p(n)}, \ldots, Pos_{t,p(n)}) \\ \wedge \bigwedge_{i \in \{-p(n),\ldots,p(n)\}} \mathit{exactlyOne}(Tape_{t,i,b_1}, \ldots, Tape_{t,i,b_l}) \end{pmatrix}$$

- *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 $w = a_1 \cdots a_n$ steht auf dem Band und alle anderen Zellen enthalten das Blank-Symbol. Dies leistet die Formel

$$\Big(\bigwedge_{i\in\{0,\ldots,n-1\}} Tape_{0,i,a_{i+1}}\Big) \wedge \Big(\bigwedge_{i\in\{-p(n),\ldots,-1\}} Tape_{0,i,\square}\Big) \wedge \Big(\bigwedge_{i\in\{n,\ldots,p(n)\}} Tape_{0,i,\square}\Big)$$

Daher setzen wir

$$Anfang := State_{0,z_0} \wedge Pos_{0,0} \wedge \Big(\bigwedge_{i\in\{0,\ldots,n-1\}} Tape_{0,i,a_{i+1}}\Big) \wedge \Big(\bigwedge_{i\in\{-p(n),\ldots,-1\}} Tape_{0,i,\square}\Big) \wedge \Big(\bigwedge_{i\in\{n,\ldots,p(n)\}} Tape_{0,i,\square}\Big)$$

- *Übergang* 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:

$$\bigwedge_{\substack{t\in\{0,\ldots,p(n)-1\},\\ z\in Z,\\ i\in\{-p(n)+1,\ldots,p(n)-1\},\\ b\in\Gamma}} \left(\begin{array}{l} \Big(State_{t,z} \wedge Pos_{t,i} \wedge Tape_{t,i,b}\Big) \\ \qquad \Longrightarrow \bigvee_{(z',b',y)\in\delta(z,b)} \Big(State_{t+1,z'} \wedge Pos_{t+1,i+dir(y)} \wedge Tape_{t+1,i,b'}\Big) \end{array} \right)$$

  - Zellen des Bandes auf denen der Kopf nicht steht, bleiben unverändert. Das leistet die Formel:

$$\bigwedge_{\substack{t\in\{0,\ldots,p(n)-1\},\\ i\in\{-p(n),\ldots,p(n)\},\\ b\in\Gamma}} \Big(\Big(\neg Pos_{t,i} \wedge Tape_{t,i,b}\Big) \implies Tape_{t+1,i,b}\Big)$$

Insgesamt ergibt sich daher

$$\begin{aligned} \ddot{U}bergang := \Bigg(& \bigwedge_{\substack{t\in\{0,\ldots,p(n)-1\},\\ z\in Z,\\ i\in\{-p(n)+1,\ldots,p(n)-1\},\\ b\in\Gamma}} \left(\begin{array}{l} \Big(State_{t,z} \wedge Pos_{t,i} \wedge Tape_{t,i,b}\Big) \\ \qquad \Longrightarrow \bigvee_{(z',b',y)\in\delta(z,b)} \Big(State_{t+1,z'} \wedge Pos_{t+1,i+dir(y)} \wedge Tape_{t+1,i,b'}\Big) \end{array} \right) \Bigg) \\ \wedge \Bigg(& \bigwedge_{\substack{t\in\{0,\ldots,p(n)-1\},\\ i\in\{-p(n),\ldots,p(n)\},\\ b\in\Gamma}} \Big(\Big(\neg Pos_{t,i} \wedge Tape_{t,i,b}\Big) \implies Tape_{t+1,i,b}\Big)\Bigg) \end{aligned}$$

- *Ende* beschreibt die Endbedingung, dass ein akzeptierender Zustand erreicht wird:

$$Ende := \bigvee_{z\in E, t\in\{0,\ldots,p(n)\}} State_{t,z}$$

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 *Übergang*-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 $I(\ddot{U}bergang) = 1$, da alle gemachten Übergänge auch genau in den Formeln abgebildet sind. Schließlich wurde durch die erzeugte Belegung für *Übergang* auch die Variable $State_{t_e,z_e}$ 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 15.2.3 an: Daher hat jede der *exactlyOne*$(\ldots)$-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*$(\ldots)$-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 *Übergang*-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. □

Satz 15.2.5 (Satz von Cook)**.** *Das Erfüllbarkeitsproblem der Aussagenlogik ist $\mathcal{NP}$-vollständig.*

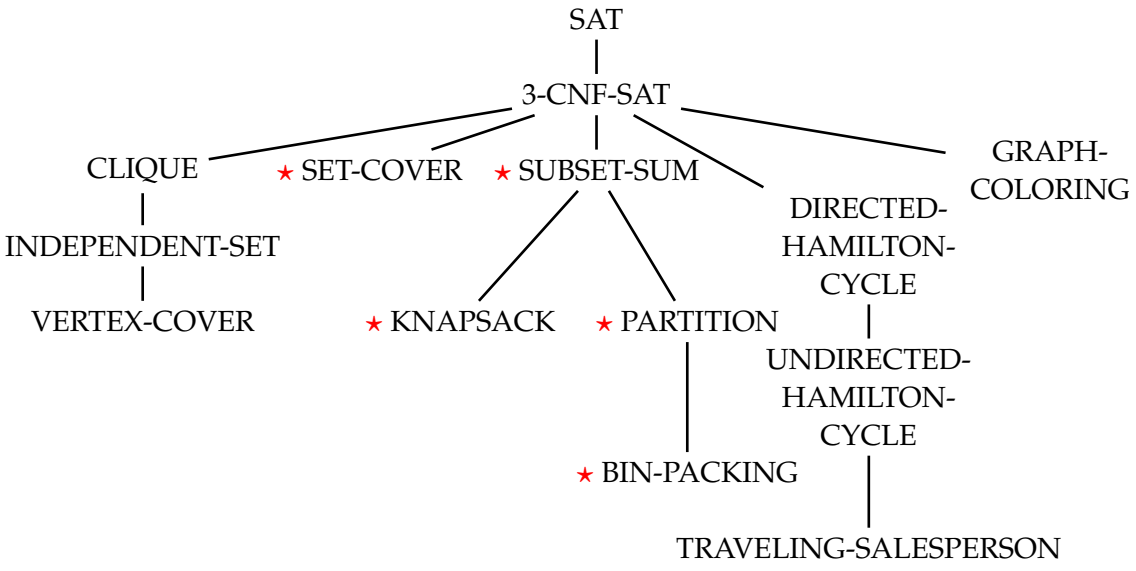
Beweis. Dies gilt, da SAT $\in \mathcal{NP}$ (gezeigt in Lemma 15.2.2) und SAT $\mathcal{NP}$-schwer ist (gezeigt in Lemma 15.2.4). □

16. Eine Auswahl NP-vollständiger Probleme

In diesem Kapitel 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.

Das folgende Diagramm zeigt unser Vorgehen: Es zeigt die in diesem Kapitel betrachteten Probleme und die Polynomialzeitreduktionen die wir präsentieren werden:



Für die in diesem Kapitel betrachteten Graphenprobleme nehmen wir stets an, dass der Graph schlingenfrei ist (d.h. für jede Kante $\{u, v\}$ eines Graphen gilt stets $u \neq v$).

16.1. Das 3-CNF-SAT-Problem

Definition 16.1.1 (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 (3-CNF)*
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 Teilformeln $\bigvee_{j=1}^{n_i} L_{i,j}$ bezeichnet man dabei als Klausel. Oft schreibt man eine konjunktive Normalform als Menge von Mengen: $\{\{L_{1,1}, \ldots, L_{1,n_1}\}, \ldots, \{L_{m,1}, \ldots, L_{m,n_m}\}\}$.

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.

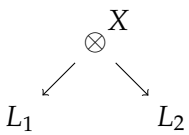
Beachte, dass eine erfüllende Belegung einer CNF, mindestens ein Literal pro Klausel wahr machen muss.

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 16.1.2. *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 nichtdeterministisch 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 SAT $\leq_{\mathrm{p}}$ 3-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 ist. Die Transformation muss daher nicht eine äquivalente Formel, sondern nur eine erfüllbarkeitsäquivalente Formel liefern.
  - 1. Schritt: Schiebe alle Negationen nach innen vor die Variablen. Dafür werden die Regeln $\neg\neg F \to F$, $\neg(F \wedge G) \to \neg F \vee \neg G$, $\neg(F \vee G) \to \neg F \wedge \neg G$, $\neg(F \iff G) \to (\neg F) \iff G$ und $\neg(F \implies G) \to F \wedge \neg G$ 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

    die Formel $X \iff L_1 \otimes L_2$, wobei $\otimes \in \{\iff, \wedge, \vee, \implies\}$ und L_1 bzw. L_2 entweder eine aussagenlogische Variable (für Nichtblatt-Knoten) oder das Literal am Blatt ist.

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 Teilformel $(X_i \iff (X_j \otimes_i X_k))$ die CNF mit dem üblichen Algorithmus. Die Größe dieser Teilformel 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.

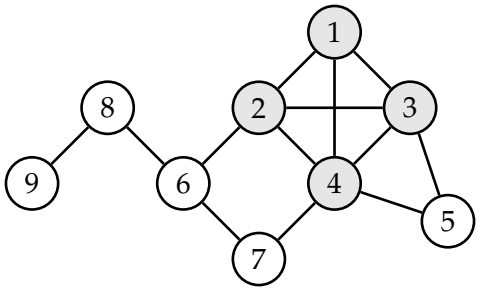
Die Größe der 3-CNF ist polynomiell in der ursprünglichen Formel und die 3-CNF kann in Polynomialzeit berechnet werden. Sei f die Funktion, die aus der aussagenlogischen Formel F diese 3-CNF berechnet. Dann gilt F erfüllbar g.d.w. $f(F)$ erfüllbar: Wenn es eine Belegung I gibt, sodass $I(F) = 1$ gilt, dann kann I erweitert werden zu I', sodass $I'(f(F)) = 1$: Setze $I'(X) = I(X)$ für alle Variablen, die in F vorkommen. Für alle 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 SAT $\leq_p$ 3-CNF-SAT. Da SAT $\mathcal{NP}$-schwer, ist somit auch 3-CNF-SAT $\mathcal{NP}$-schwer. □

16.2. 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 16.2.1. *Der Graph $G = (V, E)$ mit Knotenmenge $V = \{1, \ldots, 9\}$ 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 16.2.2 (CLIQUE-Problem)**.** *Das CLIQUE-Problem lässt sich in der gegeben/gefragt-Notation formulieren durch:*

gegeben: *Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$*
gefragt: *Besitzt G eine Clique der Größe mindestens k?*

Satz 16.2.3. *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-CNF-SAT $\leq_{\mathrm{p}}$ 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 $F = K_1 \wedge \ldots \wedge K_m$ eine 3-CNF, wobei $K_i = (L_{i,1} \vee L_{i,2} \vee L_{i,3})$ für $i = 1, \ldots, m$. 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. $V = \{(i,j) \mid i \in \{1,\ldots,m\} \text{ und } j \in \{1,2,3\}\}$. Für die Kantenmenge ziehen wir keine Kante innerhalb der drei Knoten, die aus einer Klausel stammen, sondern nur Kanten zwischen „verschiedenen Klauseln“, d.h. $E \subseteq \{\{(i,j),(i',j')\} \mid i \neq i' \wedge i,i' \in \{1,\ldots,m\} \wedge j,j' \in \{1,2,3\}\}$. 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: $\overline{x} = \neg x$ und $\overline{\neg x} = x$).

  Das ergibt $E := \{\{(i,j),(i',j')\} \mid i \neq i' \wedge i,i' \in \{1,\ldots,m\} \wedge j,j' \in \{1,2,3\}, L_{i,j} \neq \overline{L_{i',j'}}\}$. 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 $L_{1,j_1}, \ldots, L_{m,j_m}$ mit $I(L_{1,j_1}) = 1, \ldots, I(L_{m,j_m}) = 1$. 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 $V' = \{(i_1,j_1),\ldots(i_m,j_m)\}$ geben, die eine Clique der Größe m formen. Da Knoten (i,x) und (i,y) nie miteinander verbunden sind, müssen alle $i_1,\ldots,i_m$ paarweise verschieden sein, also $\{i_1,\ldots,i_m\} = \{1,\ldots,m\}$ gelten. Damit folgt: Die Literale $L_{i_1,j_1},\ldots L_{i_m,j_m}$ 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 $I(F) = 1$. □

Beispiel 16.2.4. *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. 16.1 dargestellt gezeichnet werden. Z.B. ist $\{(1,1),(2,2),(3,3)\}$ eine Clique der Größe 3. Die zugehörige Belegung ist $I(x_1) = 1, I(x_2) = 1, I(x_3) = 0$.

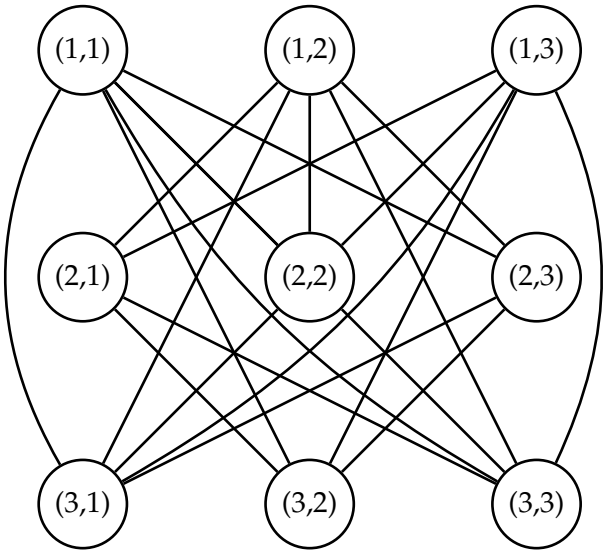
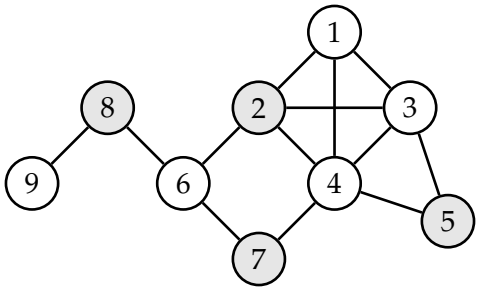


Abbildung 16.1.: Graph zu Beispiel Beispiel 16.2.4

16.3. 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 16.3.1. *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 $\overline{G} = (V, \overline{E})$ mit $\overline{E} = \{\{u, v\} \mid u, v \in V, u \neq v, \{u, v\} \notin E\}$.

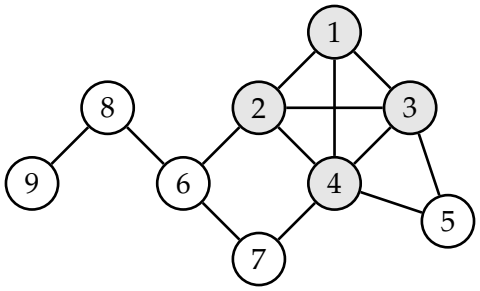
Lemma 16.3.2. *Für jeden ungerichteten Graph G gilt: G hat eine unabhängige Knotenmenge der Größe k g.d.w.* $\overline{G}$ *eine Clique der Größe k hat.*

Beweis. Sei $G = (V, E)$. Sei $V' \subseteq V$. Dann gilt:

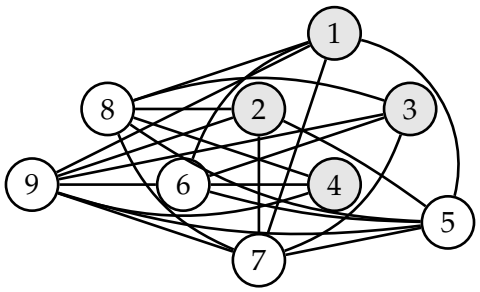
	V' ist unabhängige Knotenmenge der Größe k
g.d.w.	$\{u, v\} \notin E$ und $\lvert V' \rvert = k$ für alle $u, v \in V'$
g.d.w.	$\{u, v\} \in \overline{E}$ für alle $u, v \in V'$ und $u \neq v$
g.d.w.	V' ist eine Clique der Größe k in $\overline{G}$

□

Beispiel 16.3.3. *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 16.3.4 (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* $k \in \mathbb{N}$
gefragt: *Besitzt* G *eine unabhängige Knotenmenge der Größe mindestens* k*?*

Satz 16.3.5. *INDEPENDENT-SET ist* $\mathcal{NP}$*-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 $\mathcal{NP}$-schwer. Wir zeigen CLIQUE $\leq_{\mathrm{p}}$ INDEPENDENT-SET. Sei $f((V, E, m)) = (V, \overline{E}, m)$ wobei $\overline{E} = \{\{u, v\} \mid u, v \in V, \{u, v\} \notin E\}$. Dann gilt (V, E) hat eine Clique der Größe m g.d.w. $(V, \overline{E})$ eine unabhängige Knotenmenge der Größe m hat. Da die Funktion f in Polynomialzeit berechnet werden kann, gilt CLIQUE $\leq_{\mathrm{p}}$ INDEPENDENT-SET und da CLIQUE $\mathcal{NP}$-schwer, folgt auch INDEPENDENT-SET ist $\mathcal{NP}$-schwer. □

16.4. 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 $u, v \in V : \{u, v\} \in E \implies u \in V' \vee v \in V'$.

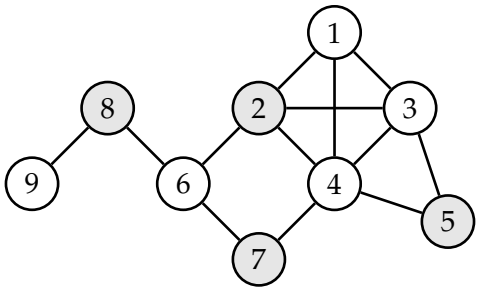
Lemma 16.4.1. *$G = (V, E)$ hat eine unabhängige Knotenmenge der Größe k g.d.w. G hat eine überdeckende Knotenmenge der Größe $|V| - k$.*

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

□

Beispiel 16.4.2. *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 16.4.3 (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 $k \in \mathbb{N}$*
gefragt:	*Besitzt G eine überdeckende Knotenmenge der Größe höchstens k?*

Satz 16.4.4. *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. □

16.5. ⋆ Das SET-COVER-Problem

Definition 16.5.1 (SET-COVER-Problem)**.** *Das SET-COVER-Problem lässt sich in der gegeben/gefragt-Notation formulieren durch:*

gegeben: *Ein System von Mengen $T_1,\ldots,T_k$ mit $T_1,\ldots,T_k \subseteq M$, wobei M eine endliche Grundmenge ist und eine Zahl $n \leq k$*
gefragt: *Gibt es eine Auswahl von n Mengen $T_{i_1},\ldots,T_{i_n}$ ($i_j \in \{1,\ldots,k\}$) mit $T_{i_1} \cup \ldots \cup T_{i_n} = M$?*

Beispiel 16.5.2. *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 16.5.3. *SET-COVER ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen die beiden benötigten Teile:

- SET-COVER $\in \mathcal{NP}$: 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.
- SET-COVER ist $\mathcal{NP}$-schwer. Wir zeigen 3-CNF-SAT $\leq_p$ SET-COVER. Sei $F = K_1 \wedge \ldots \wedge K_m$ eine 3-CNF. Seien $\{x_1,\ldots,x_n\}$ die aussagenlogischen Variablen, die in F vorkommen. Setze $M = \{1,\ldots,m+n\}$. Für $i = 1,\ldots,n$ sei
$$P_i = \{j \mid \text{Literal } x_i \text{ kommt in Klausel } K_j \text{ vor}\} \cup \{m+i\}$$
$$N_i = \{j \mid \text{Literal } \neg x_i \text{ kommt in Klausel } K_j \text{ vor}\} \cup \{m+i\}$$
Das Mengensystem sei $P_1,\ldots,P_n,N_1,\ldots,N_n \subseteq M$.

  Wenn F erfüllbar, dann hat SET-COVER eine Lösung: Sei I eine Belegung mit $I(F) = 1$. Wenn $I(x_i) = 1$, dann wähle die Menge P_i und wenn $I(x_i) = 0$, dann wähle die Menge N_i. 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 $1,\ldots,m$ vor, da jede Variable mit 0 oder 1 belegt wurde, kommt jede Zahl $m+1,\ldots,m+n$ vor.

  Umgekehrt gilt: Wenn es n Mengen $U_1,\ldots,U_n \subseteq P_1,\ldots,P_n,N_1,\ldots,N_n$ gibt, sodass $U_1 \cup \ldots \cup U_n = M$, dann ist F erfüllbar: Damit die Zahlen $m+i$ für $i = 1,\ldots,n$ alle enthalten sind, muss P_i oder N_i für jedes $i = 1,\ldots,n$ unter den U_i sein. Daher $U_i \in \{P_i,N_i\}$ für $i = 1,\ldots,n$. Wenn wir nun $I(x_i) = 1$ setzen, wenn $U_i = P_i$ und $I(x_i) = 0$, wenn $U_i = N_i$, dann erfüllt I die Formel F: Da die Zahlen $1,\ldots,m$ alle enthalten sind in $\bigcup U_i$, wird in jeder Klausel mindestens ein Literal auf wahr gesetzt.

Sei f die Funktion die aus F, dass Mengensystem $P_1, \ldots, P_n, N_1, \ldots, N_n \subseteq M$ und n als Zahl berechnet. Dann ist f in Polynomialzeit berechenbar und hat die Eigenschaft, dass F erfüllbar ist g.d.w. $f(F)$ eine lösbare SET-COVER-Instanz ist. Damit haben wir 3-CNF-SAT $\leq_p$ SET-COVER gezeigt. □

16.6. ⋆ Das SUBSET-SUM-Problem

Definition 16.6.1 (SUBSET-SUM-Problem)**.** *Das SUBSET-SUM-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:*

gegeben: *Natürliche Zahlen* $a_1, \ldots, a_k \in \mathbb{N}$ *und* $s \in \mathbb{N}$
gefragt: *Gibt es eine Teilmenge* $I \subseteq \{1, \ldots, k\}$*, sodass* $\sum_{i \in I} a_i = s$*?*

Satz 16.6.2. *Das SUBSET-SUM-Problem ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen die beiden benötigten Teile:

- SUBSET-SUM $\in \mathcal{NP}$: Rate nichtdeterministisch eine Teilmenge $I \subseteq \{1, \ldots, k\}$ und anschließend verifiziere, dass $\sum_{i \in I} a_i = s$ gilt.
- SUBSET-SUM ist $\mathcal{NP}$-schwer: Wir geben eine Polynomialzeitreduktion von 3-CNF-SAT auf SUBSET-SUM an. Sei $F = K_1 \wedge \ldots \wedge K_m$ eine 3-CNF, wobei jede Klausel genau drei Literale habe (dies kann durch Vervielfachen von Literalen stets hergestellt werden). Seien $x_1, \ldots, x_n$ die aussagenlogischen Variablen, die in F vorkommen. Wir erzeugen $(n+m)$-stellige Zahlen t_i, f_i für $i = 1, \ldots, n$:
  - In t_i ist die i-te Stelle 1. Zudem ist für $j = 1, \ldots, m$ jede $(n+j)$-te Stelle 1, falls Klausel K_j das Literal x_i enthält.
  - In f_i ist die i-te Stelle 1. Zudem ist für $j = 1, \ldots, m$ 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 $j = 1, \ldots, m$:
  - 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 $s = \underbrace{1 \ldots 1}_{n\text{-mal}} \underbrace{4 \ldots 4}_{m\text{-mal}}$.

  Sei also $f(F) = ((t_1, \ldots, t_n, f_1, \ldots, f_n, c_1, \ldots, c_m, d_1, \ldots, d_m), s)$. 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 SUBSET-SUM-Instanz $((t_1, \ldots, t_n, f_1, \ldots, f_n, c_1, \ldots, c_m, d_1, \ldots, c_m), s)$ eine Lösung I hat, kann daraus eine erfüllende Belegung J für F konstruiert werden: Setze für $i \in I$ mit $1 \leq i \leq n$: $J(x_i) = 1$ und setze für $i \in I$ mit $n+1 \leq i \leq n+n$: $J(x_i) = 0$. Umgekehrt kann eine Belegung J für F benutzt werden, um die Indizes I der Lösung für das SUBSET-SUM-Problem zu konstruieren: I enthält den Index von t_i wenn $J(x_i) = 1$, I enthält den Index von f_i wenn $J(x_i) = 0$, 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 J erfüllende Belegung).

Damit haben wir gezeigt 3-CNF-SAT $\leq_{\mathrm{p}}$ SUBSET-SUM und aus der $\mathcal{NP}$-Schwere von 3-CNF-SAT folgt die $\mathcal{NP}$-Schwere von SUBSET-SUM. □

Beispiel 16.6.3. *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 SUBSET-SUM-Instanz*

$$\begin{array}{lll} a_1 & = t_1 & = 110 \\ a_2 & = f_1 & = 101 \\ a_3 & = c_1 & = 010 \\ a_4 & = c_2 & = 001 \\ a_5 & = d_1 & = 020 \\ a_6 & = d_2 & = 002 \\ & s & = 144 \end{array}$$

transformiert. Diese ist unlösbar (was dazu passt, dass die Formel unerfüllbar ist).

Beispiel 16.6.4. *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:*

$$\begin{array}{lll} a_1 & = t_1 & = 100010100 \\ a_2 & = t_2 & = 010011010 \\ a_3 & = t_3 & = 001000001 \\ a_4 & = t_4 & = 000100101 \\ a_5 & = f_1 & = 100001010 \\ a_6 & = f_2 & = 010000101 \\ a_7 & = f_3 & = 001001010 \\ a_8 & = f_4 & = 000110000 \end{array} \qquad \begin{array}{lll} a_9 & = c_1 & = 000010000 \\ a_{10} & = c_2 & = 000001000 \\ a_{11} & = c_3 & = 000000100 \\ a_{12} & = c_4 & = 000000010 \\ a_{13} & = c_5 & = 000000001 \\ a_{14} & = d_1 & = 000020000 \\ a_{15} & = d_2 & = 000002000 \\ a_{16} & = d_3 & = 000000200 \\ a_{17} & = d_4 & = 000000020 \\ a_{18} & = d_5 & = 000000002 \\ s & & = 111144444 \end{array}$$

Eine Lösung ist $I = \{2, 4, 5, 7, 9, 10, 11, 12, 13, 14, 16, 18\}$. *Die zugehörige erfüllende Belegung ist* $J(x_1) = 0, J(x_2) = 1, J(x_3) = 0, J(x_4) = 1$.

16.7. ⋆ Das KNAPSACK-Problem

Definition 16.7.1 (KNAPSACK-Problem)**.** *Das KNAPSACK-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:*

gegeben: *k Gegenstände mit Gewichten* $w_1, \ldots, w_k \in \mathbb{N}$ *und Nutzenwerten* $n_1, \ldots, n_k \in \mathbb{N}$, *sowie zwei Zahlen* $s_w, s_n \in \mathbb{N}$
gefragt: *Gibt es eine Teilmenge* $I \subseteq \{1, \ldots, k\}$, *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 SUBSET-SUM-Problem, es ist daher ein Spezialfall. Daher ist die Reduktion SUBSET-SUM $\leq_p$ KNAPSACK sehr einfach.

Satz 16.7.2. *KNAPSACK ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen die beiden benötigten Teile:

- KNAPSACK $\in \mathcal{NP}$: Rate eine Teilmenge $I \subseteq \{1, \ldots, k\}$ 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.
- KNAPSACK ist $\mathcal{NP}$-schwer: Sei $((a_1, \ldots, a_k), s)$ eine SUBSET-SUM-Instanz, dann sei $f((a_1, \ldots, a_k), s) = ((w_1, \ldots, w_k), (n_1, \ldots, n_k), s_w, s_m)$ mit $w_i = a_i, n_i = a_i$ für $i = 1, \ldots, k$ und $s_w = s$ und $s_m = s$. Es gilt $((a_1, \ldots, a_k), s)$ hat eine Lösung g.d.w. $f((a_1, \ldots, a_k), s)$ hat eine Lösung und f ist in polynomieller Zeit von einer DTM berechenbar. Daher gilt SUBSET-SUM $\leq_p$ KNAPSACK. Damit folgt aus der $\mathcal{NP}$-Schwere von SUBSET-SUM auch die $\mathcal{NP}$-Schwere von KNAPSACK. □

16.8. ⋆ Das PARTITION-Problem

Definition 16.8.1 (PARTITION-Problem)**.** *Das PARTITION-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:*

gegeben: *Natürliche Zahlen* $a_1, \ldots, a_k \in \mathbb{N}$
gefragt: *Gibt es eine Teilmenge* $I \subseteq \{1, \ldots, k\}$ *sodass* $\sum_{i \in I} a_i = \sum_{i \in \{1,\ldots,k\} \setminus I} a_i$*?*

Satz 16.8.2. *PARTITION ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen die beiden benötigten Teile:

- PARTITION $\in \mathcal{NP}$: Rate nichtdeterministisch $I \subseteq \{1, \ldots, k\}$ und prüfe anschließend, ob $\sum_{i \in I} a_i = \sum_{i \in \{1,\ldots,k\} \setminus I} a_i$ gilt. Die dazu passende NTM läuft in (nichtdeterministischer) Polynomialzeit.
- PARTITION ist $\mathcal{NP}$-schwer: Wir zeigen SUBSET-SUM $\leq_p$ PARTITION. Sei $f((a_1, \ldots, a_k), s) = (a_1, \ldots, a_k, a_{k+1}, a_{k+2})$ mit $a_{k+1} = A + s$ und $a_{k+2} = 2A - s$, wobei $A = \sum_{i=1}^{k} a_k$. Die Funktion f kann von einer DTM in polynomieller Zeit berechnet werden.

  Wenn $(a_1, \ldots, a_k, a_{k+1}, a_{k+2})$ eine PARTITION-Lösung $I \subseteq \{1, \ldots, k+2\}$ hat, dann gilt $\sum_{i \in I} a_i = 2A = \sum_{i \in \{1,\ldots,k+2\} \setminus I} a_i$. 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 SUBSET-SUM-Instanz $((a_1, \ldots, a_k), s)$. Wenn $k+1 \in I$, dann sei $I' = \{1, \ldots, k\} \setminus I$ und damit $\sum_{i\in I'} = \sum_{i\in\{1,\ldots,k+2\}\setminus I} - a_{k+2} = 2A - (2A - s) = s$ und I' ist eine Lösung für die SUBSET-SUM-Instanz $((a_1, \ldots, a_k), s)$.

Wenn $I \subseteq \{1, \ldots, k\}$ eine Lösung für die SUBSET-SUM-Instanz $((a_1, \ldots, a_k), s)$, dann ist $I \cup \{k+2\}$ eine Lösung für die PARTITION-Instanz $(a_1, \ldots, a_k, a_{k+1}, a_{k+2})$, weil $\sum_{I\cup\{k+2\}} a_i = s + (2A - s) = 2A$, was die Hälfte von $\sum_{i=1}^{k+2} = 4A$ ausmacht.

Daher gilt $((a_1, \ldots, a_k), s)$ ist lösbar g.d.w. $f((a_1, \ldots, a_k), s)$ ist lösbar. Damit haben wir SUBSET-SUM $\leq_p$ PARTITION gezeigt. Da SUBSET-SUM $\mathcal{NP}$-schwer ist, ist auch PARTITION $\mathcal{NP}$-schwer. □

Beispiel 16.8.3. *Sei ((1,2,3,4,5,6),14) eine SUBSET-SUM-Instanz. Dann ist (1,2,3,4,5,6,35,28) die von der Funktion f erzeugte PARTITION-Instanz. Eine Lösung ist $I = \{1, 3, 4, 6, 28\}$ da $1 + 3 + 4 + 6 + 28 = 42 = 2 + 5 + 35$. Damit ist $I' = \{1, 3, 4, 6\}$ eine Lösung der SUBSET-SUM-Instanz (was auch stimmt, da $1 + 3 + 4 + 6 = 14$).*

16.9. ⋆ Das BIN-PACKING-Problem

Definition 16.9.1 (BIN-PACKING-Problem)**.** *Das BIN-PACKING-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:*

gegeben: *Natürliche Zahlen $a_1, \ldots, a_k \in \mathbb{N}$, die Behältergröße $b \in \mathbb{N}$ 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 assign : $\{1, \ldots, k\} \to \{1, \ldots, m\}$, sodass für alle $j \in \{1, \ldots, m\}$ gilt: $\sum_{assign(i)=j} a_i \leq b$?*

Satz 16.9.2. *BIN-PACKING ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen die beiden benötigten Teile:

- BIN-PACKING $\in \mathcal{NP}$: Rate nichtdeterministisch für jede Zahl a_i in welchen Behälter sie gehört. (D.h. rate die Funktion *assign*.) Anschließend verifiziere, dass die geratene Funktion eine Lösung ist. Dies kann in Polynomialzeit auf einer NTM durchgeführt werden.
- BIN-PACKING ist $\mathcal{NP}$-schwer. Sei $(a_1, \ldots, a_k)$ eine PARTITION-Instanz. Dann sei $f(a_1, \ldots, a_k)$ die BIN-PACKING-Instanz mit Zahlen $a_1, \ldots, a_k$, 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 BIN-PACKING-Instanz ebenso (da die Behälter nicht ausreichen). Anderenfalls sei $I \subseteq \{1, \ldots, k\}$ eine Lösung für die PARTITION-Instanz, dann ist

$$assign(i) = \begin{cases} 1, & \text{wenn } i \in I \\ 2, & \text{wenn } i \notin I \end{cases}$$

  eine Lösung für die BIN-PACKING-Instanz. Umgekehrt kann aus einer Lösung für die BIN-PACKING-Instanz mit $I = \{i \mid 1 \leq i \leq k, assign(i) = 1\}$ eine Lösung für die PARTITION-Instanz erstellt werden. Da f in Polynomialzeit berechnet werden kann, gilt

PARTITION $\leq_p$ BIN-PACKING. Aus der $\mathcal{NP}$-Schwere von PARTITION folgt die $\mathcal{NP}$-Schwere von BIN-PACKING. □

16.10. Das DIRECTED-HAMILTON-CYCLE-Problem

Definition 16.10.1 (DIRECTED-HAMILTON-CYCLE-Problem)**.** *Das DIRECTED-HAMILTON-CYCLE-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:*

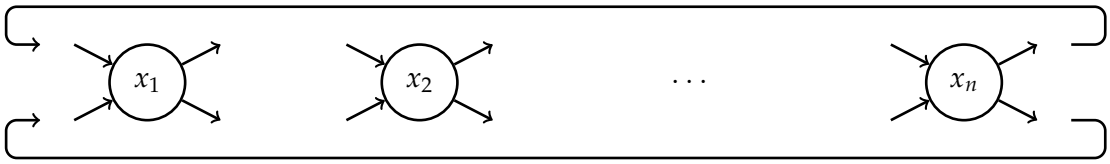
gegeben: *Ein gerichteter Graph $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$*
gefragt: *Gibt es einen Hamilton-Kreis in G, d.h. einen Kreis, der genau alle Knoten einmal besucht? Formaler kann dies als Frage formuliert werden: Gibt es eine Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, sodass $(v_{\pi(i)}, v_{\pi(i+1)}) \in E$ und $(v_{\pi(n)}, v_{\pi(1)}) \in E$?*

Satz 16.10.2. *DIRECTED-HAMILTON-CYCLE ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen die beiden benötigten Teile:

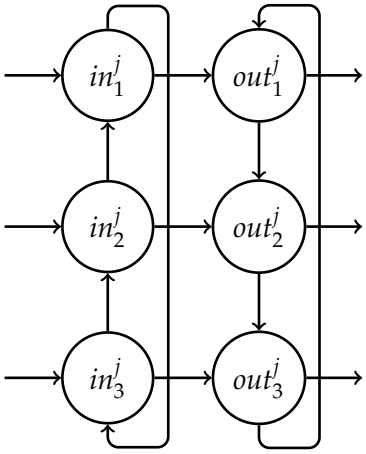
- DIRECTED-HAMILTON-CYCLE $\in \mathcal{NP}$: Rate nichtdeterministisch die Permutation π und verifiziere anschließend, ob $(v_{\pi(i)}, v_{\pi(i+1)}) \in E$ und $(v_{\pi(n)}, v_{\pi(1)}) \in E$ gilt. Dies kann auf einer NTM in Polynomialzeit entschieden werden.
- DIRECTED-HAMILTON-CYCLE ist $\mathcal{NP}$-schwer: Wir geben eine Polynomialzeitreduktion von 3-CNF-SAT auf DIRECTED-HAMILTON-CYCLE an. Sei $F = K_1 \wedge \ldots \wedge K_m$ eine 3-CNF, sodass jede Klausel K_i genau 3 Literale enthält. Seien $\{x_1, \ldots, x_n\}$ die aussagenlogischen Variablen, die in F vorkommen.

  Dann erzeugen wir zunächst einen Graph der Form



  Die Idee dabei ist, dass der Hamilton-Kreis Knoten x_i oben durchläuft, wenn x_i in der erfüllenden Belegung auf 1 gesetzt wird und x_i unten durchläuft, wenn x_i in der erfüllenden Belegung auf 0 gesetzt wird. Da nur einer der beiden Durchläufe möglich ist, ist die Belegung dadurch wohl-definiert.

  Für das Auftreten der x_i in den Klauseln betrachten wir zunächst den folgenden Teilgraphen, den wir mit K_j bezeichnen:



Dieser Teilgraph hat sehr besondere Eigenschaften:

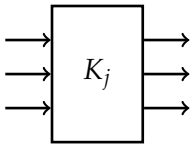
- Jeder Hamilton-Kreis, der durch den Eingang in_i^j in K_j hereingeht, muss K_j durch den Knoten out_i^j verlassen: Anderenfalls gibt es „blockierte Knoten“, die durch den Hamilton-Kreis nicht mehr besucht werden können (betrachte alle endlich vielen Fälle).
- Ein Hamilton-Kreis, der in in_1^j in K_j 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
  - $in_2^j, in_1^j, out_1^j, out_1^j$, 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 $x_1, \ldots, x_n$ erzeugt werden und die jeweils oberen und unteren Pfade von x_i und x_{i+1} direkt verbunden werden. Anschließend werden die Klauseln $K_1, \ldots, K_m$ 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.

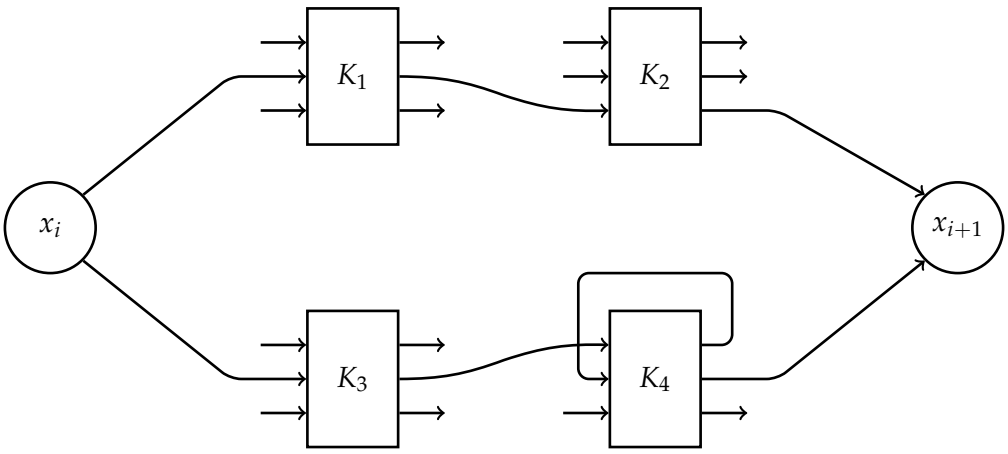
Dies zeigt: 3-CNF-SAT $\leq_p$ DIRECTED-HAMILTON-CYCLE und damit die $\mathcal{NP}$-Schwere von DIRECTED-HAMILTON-CYCLE.

□

Beispiel 16.10.3. *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



16.11. Das UNDIRECTED-HAMILTON-CYCLE-Problem

Definition 16.11.1 (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 $V = \{v_1, \ldots, v_n\}$*
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 $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, sodass $\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$ und $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$?*

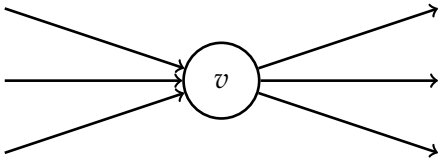
Satz 16.11.2. *UNDIRECTED-HAMILTON-CYCLE ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen die beiden benötigten Teile:

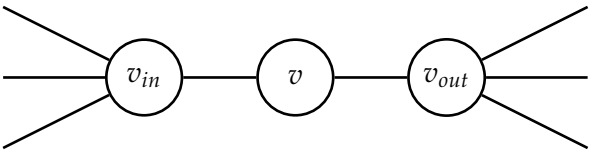
- UNDIRECTED-HAMILTON-CYCLE $\in \mathcal{NP}$: Rate die Permutation π nichtdeterministisch und verifiziere anschließend, ob $\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$ und $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$ gilt. Dies kann auf einer NTM in Polynomialzeit durchgeführt werden.
- UNDIRECTED-HAMILTON-CYCLE ist $\mathcal{NP}$-schwer. Wir zeigen DIRECTED-HAMILTON-CYCLE $\leq_{\mathrm{p}}$ UNDIRECTED-HAMILTON-CYCLE. Sei f die Funktion, die aus einem gerichteten Graphen (V, E) einen ungerichteten Graphen macht, sodass

$$\begin{aligned} f(V) &= \bigcup\{\{v_{in}, v, v_{out}\} \mid v \in V\} \\ f(E) &= \{\{u_{out}, v_{in}\} \mid (u, v) \in E\} \cup \bigcup\{\{\{v_{in}, v\}, \{v, v_{out}\}\} \mid v \in V\} \end{aligned}$$

  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

$$v_{in,\pi(1)}, v_{\pi(1)}, v_{out,\pi(1)}, \ldots v_{in,\pi(n)}, v_{\pi(n)}, v_{out,\pi(n)}, v_{in,\pi(1)}$$

  ein ungerichteter Hamilton-Kreis, dann ist $v_{\pi(1)}, \ldots, v_{\pi(n)}, v_{\pi(1)}$ der gerichtete Hamilton Kreis; und bei ungerichtetem Hamilton-Kreis

$$v_{out,\pi(1)}, v_{\pi(1)}, v_{in,\pi(1)}, \ldots v_{out,\pi(n)}, v_{\pi(n)}, v_{in,\pi(n)}, v_{out,\pi(1)}$$

gibt es den gerichteten Hamilton Kreis $v_{\pi(n)}, \ldots, v_{\pi(1)}, v_{\pi(n)}$.

Auch umgekehrt ist klar, dass es für jeden gerichteten Hamilton-Kreis einen gerichteten gibt. Da f in Polynomialzeit berechenbar ist, haben zusammenfassend gezeigt, dass DIRECTED-HAMILTON-CYCLE $\leq_p$ UNDIRECTED-HAMILTON-CYCLE gilt. □

16.12. Das TRAVELING-SALESPERSON-Problem

Definition 16.12.1 (TRAVELING-SALESPERSON-Problem)**.** *Das TRAVELING-SALESPERSON-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:*

gegeben: *Eine Menge von Knoten (Städten) $V = \{v_1, \ldots, v_n\}$, eine $(n \times n)$-Matrix $(M_{i,j})$ Entfernungen $M_{i,j} \in \mathbb{N}$ zwischen den Städten v_i und v_j, sowie eine Zahl $k \in \mathbb{N}$*

gefragt: *Gibt es eine Rundreise der Länge k oder weniger, die alle Städte besucht und der Startort gleich dem Zielort ist?*
Formal: Gibt es eine Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, sodass $(\sum_{i=1}^{n-1} M_{\pi(i),\pi(i+1)}) + M_{\pi(n)+\pi(1)} \leq k$?

Satz 16.12.2. *Das TRAVELING-SALESPERSON-Problem ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen die beiden benötigten Teile:

- TRAVELING-SALESPERSON $\in \mathcal{NP}$: Rate die Permutation π 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.
- TRAVELING-SALESPERSON ist $\mathcal{NP}$-schwer: Wir reduzieren UNDIRECTED-HAMILTON-CYCLE in Polynomialzeit auf TRAVELING-SALESPERSON: Sei $G = (V, E)$ mit $V = \{1, \ldots, n\}$. 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-CYLE $\leq_p$ TRAVELING-SALESPERSON.

□

16.13. Das GRAPH-COLORING-Problem

Definition 16.13.1 (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 $k \in \mathbb{N}$*

gefragt: *Gibt es Färbung der Knoten in V mit $\leq$ k Farben, sodass keine zwei benachbarten Knoten in G die gleiche Farbe erhalten?*

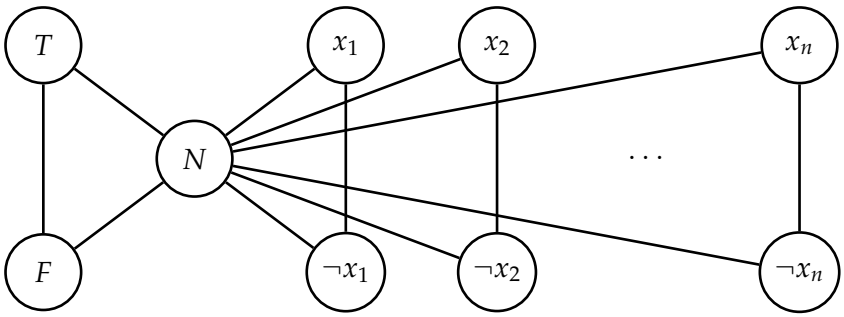
Satz 16.13.2. *GRAPH-COLORING ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen die beiden benötigten Teile:

- GRAPH-COLORING $\in \mathcal{NP}$: Rate nichtdeterministisch die Farbe aus $\{1, \ldots, k\}$ 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 $\leq_{\mathrm{p}}$ GRAPH-COLORING. Sei $F = K_1 \wedge \ldots \wedge K_m$ eine 3-CNF, sodass jede Klausel K_i genau 3 Literale enthält. Seien $\{x_1, \ldots, x_n\}$ die aussagenlogischen Variablen, die in F vorkommen.

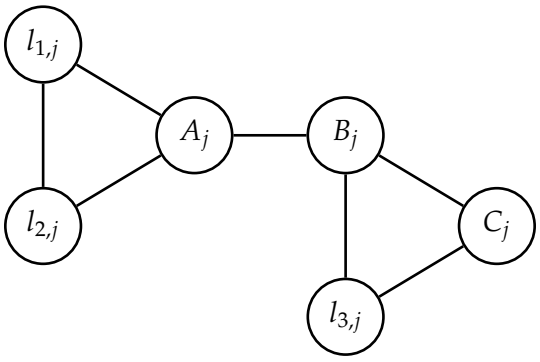
  Wir erzeugen ein GRAPH-COLORING Problem mit $k = 3$, d.h. der Graph muss mit drei Farben färbbar sein.

  Konstruiere zunächst den Teilgraphen

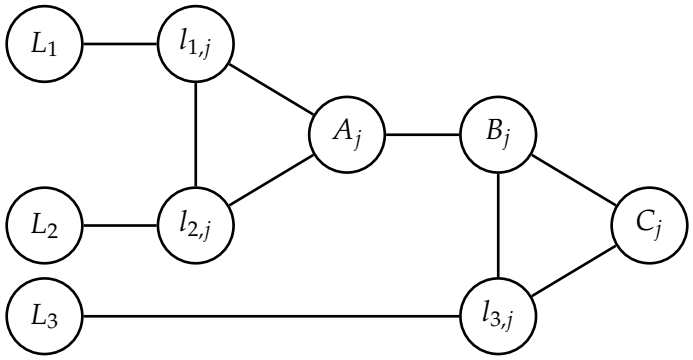


Wenn der Graph 3-färbbar ist, dann müssen T, N, F mit allen drei Farben gefärbt werden. O.B.d.A. seien dies genau die Farben T, N, F und die Knoten entsprechend gefärbt (T mit T, N mit N, F mit F). Dann muss für eine Färbung stets $(x_i, \neg x_i)$ mit (T, F) oder (F, T) gefärbt werden. D.h. dieser Teilgraph 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 $K_1, \ldots, K_m$ 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-CNF-SAT $\leq_{\mathrm{p}}$ GRAPH-COLORING gezeigt. □

Literatur

ACKERMANN, Wilhelm: Zum Hilbertschen Aufbau der reellen Zahlen. In: *Mathematische Annalen* 99 (1928), Dec, Nr. 1, S. 118–133

COCKE, John: *Programming Languages and Their Compilers: Preliminary Notes*. 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*, 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)*. 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, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015 (Leibniz International Proceedings in Informatics (LIPIcs)). – ISSN 1868–8969, 649–661

OGDEN, William F.: A Helpful Result for Proving Inherent Ambiguity. In: *Mathematical Systems Theory* 2 (1968), Nr. 3, S. 191–194

PÉTER, Rózsa: Konstruktion nichtrekursiver Funktionen. In: *Mathematische Annalen* 111 (1935), Dec, Nr. 1, S. 42–60

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

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