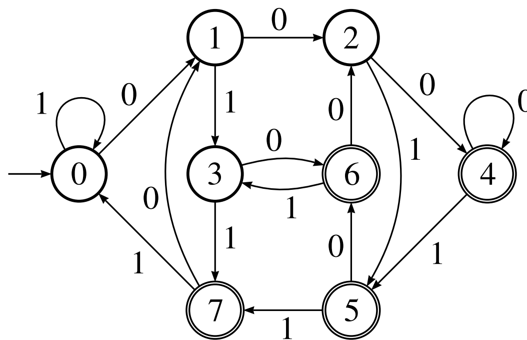
Automatentheorie
und
Formale Sprachen

Eine Einführung in Automatentheorie,
formale Sprachen, Berechenbarkeit und Komplexität

Peter Barth
Hochschule RheinMain, Wiesbaden

6. November 2019



Inhaltsverzeichnis

Übersicht **3**

1 Endliche Automaten **5**
1.1 Deterministische endliche Automaten 7
1.2 Modell der Arbeitsweise eines Automaten 9
1.3 Nicht-deterministische endliche Automaten 10
1.4 Endliche Automaten mit ε-Übergängen 15
1.5 Äquivalenz und Minimierung von Automaten 17
1.6 Anwendung in der Texterkennung 20

2 Reguläre Sprachen **26**
2.1 Reguläre Ausdrücke und Sprachen 26
2.2 Endliche Automaten und reguläre Sprachen 27
2.3 Weitere Operationen und Abschlusseigenschaften regulärer Sprachen . . 33
2.4 Zwei Beispiele für nicht reguläre Sprachen 34
2.5 Das Pumping-Lemma für reguläre Sprachen 34

3 Grammatiken formaler Sprachen **36**
3.1 Semi-Thue-Systeme 36
3.2 Chomsky-Grammatiken 37
3.3 Die Chomsky-Hierarchie 38
3.4 Endliche Automaten und rechtslineare Grammatiken 39

4 Kontextfreie Sprachen **42**
4.1 Mehrdeutigkeit bei kontextfreien Grammatiken 44
4.2 Normalformen kontextfreier Grammatiken 47
4.3 Das Pumping-Lemma für kontextfreie Sprachen 53

5 Kellerautomaten und kontextfreie Sprachen **55**
5.1 Allgemeines zu Kellerautomaten 55
5.2 Deterministische Kellerautomaten 57
5.3 Nicht-Deterministische Kellerautomaten 59

5.4 Kellerautomaten und kontextfreie Grammatiken 60
5.5 Endliche Automaten und Kellerautomaten 60
5.6 Alternativen für das Akzeptieren eines Wortes 61

6 Das Problem der Syntaxanalyse **63**
6.1 Das CYK-Verfahren 63
6.2 Syntaxanalyse mit LL(k)-Grammatiken 65

7 Allgemeinere Chomsky-Sprachen **69**
7.1 Sprachen vom Chomsky-Typ 1 69
7.2 Sprachen vom Chomsky-Typ 0 70

8 Turing-Maschinen und allgemeinere Chomsky-Sprachen **72**
8.1 Turing-Maschinen 72
8.2 Turing-Maschinen als Akzeptoren 75
8.3 Erweiterungen der Turing-Maschine 76
8.4 Linear beschränkte Automaten 77

9 Entscheidbarkeit und Berechenbarkeit **79**
9.1 Nicht entscheidbare Probleme 81
9.2 Nicht entscheidbare Sprachen 83
9.3 Das Halteproblem für Turing-Maschinen 85
9.4 Die nicht berechenbare fleißige Biber Funktion 86

10 Nicht handhabbare Probleme **89**
10.1 Laufzeit, Komplexität und Problemgröße 89
10.2 Die Problemklassen *P* und *NP* 91
10.3 *NP*-vollständige Probleme 93

Übersicht

Die theoretische Informatik befasst sich unter anderem mit den formalen Grundlagen von Berechnungen auf Computern. Unabhängig von spezifischer Hardware werden einfache mathematische Modelle (abstrakte Maschinen) untersucht, die es uns erlauben Berechnungen beziehungsweise Programmausführungen zu untersuchen. Die theoretische Informatik gibt uns Antworten auf die folgenden Fragen:

- Was ist mit welcher Art von Maschinen berechenbar?
- Welche Arten von Berechnungsmodellen gibt es und wie unterscheiden sie sich?
- Was ist überhaupt berechenbar?
- Wie lange dauert es etwas zu berechnen?

Für diese Aussagen sind die Automatentheorie, formale Sprachen & Grammatiken sowie Berechenbarkeit & Komplexität die wichtigsten Teilgebiete. Schnelles Suchen in Texten, Syntaxanalyse und Compilerbau sind direkte Anwendungsbereiche der theoretischen Grundlagen.

Viele Fragestellungen lassen sich zurückführen auf Aussagen über Wörter mit Zeichen aus einem Alphabet. Eine abstrakte Maschine soll zum Beispiel feststellen, ob ein Wort eine spezielle Eigenschaft hat (*Erkennen*). Ein anderes Berechnungsmodell generiert Wörter mit dieser speziellen Eigenschaft (*Erzeugen*). Alle Worte mit einer speziellen Eigenschaft heißen *Sprache*.

Wir werden einige dieser abstrakten Maschinen und Berechnungsmodelle kennen lernen. Dabei werden wir sehen, dass es Unterschiede in der *Mächtigkeit* verschiedener Modelle gibt, aber auch, dass auf den ersten Blick unterschiedliche Modelle gleich mächtig sind. Nicht alle Sprachen können also von allen Modellen erkannt oder erzeugt werden. Wir werden diese Sprachen klassifizieren und jeder Klasse ein abstraktes Modell zum Erzeugen und Erkennen von Sprachen zuordnen. Eine Klassifikation von Sprachen wurde von *Chomsky* entwickelt.

In der Übersichtstabelle finden Sie die Chomsky-Hierarchie sowie dazu passend einige abstrakten Maschinen und Berechnungsmodelle zum Erkennen beziehungsweise Erzeugen von Sprachen der jeweiligen Sprachklasse. Für Sprachen vom Typ 0, die so mächtig sind wie jeder vorstellbare Computer, wird untersucht was überhaupt berechenbar ist und

ÜBERSICHTSTABELLE

<table>
<tr><th>Sprachen
Chomsky-Hierarchie
(Kapitel 3)</th><th>Erkennen
Abstrakte Maschinen</th><th>Erzeugen
Grammatiken (Kapitel 3)</th></tr>
<tr><td>Typ 3,
regulär</td><td>Endliche Automaten
(Kapitel 1)</td><td>Rechtslineare Grammatiken,
reguläre Ausdrücke
(Kapitel 2, 3.4)</td></tr>
<tr><td>Typ 2,
kontextfrei</td><td>Kellerautomaten
(Kapitel 5, 6)</td><td>Kontextfreie Grammatiken
(Kapitel 4)</td></tr>
<tr><td rowspan="2">Typ 1,
kontext-sensitiv
Typ 0
(Kapitel 7)</td><td>Turing-Maschinen
(Kapitel 8)</td><td></td></tr>
<tr><td colspan="2">Berechenbarkeit (Kapitel 9)
Komplexität (Kapitel 10)</td></tr>
</table>

was in praktikabler Zeit berechenbar ist. Gleichzeitig dient die Übersichtstabelle als grobe Strukturierung der Themen dieses Skripts.

Automatentheorie, formale Sprachen & Grammatiken sowie Berechenbarkeit & Komplexität sind inzwischen gut verstandene Themen der theoretischen Informatik. Es gibt eine Reihe guter Lehrbücher, von denen einige hier aufgezählt sind.

- Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie; *John E. Hopcroft, Jeffrey D. Ullman, Rajeev Motwani*, Pearson, 2002/2011
- Theoretische Informatik – kurz gefasst; *Uwe Schöning*, Spektrum, 2008

Ich danke den Studierenden der Medieninformatik herzlich für das konstruktive Feedback zu dem vorliegenden Skript. Ihnen ist unter anderem zu verdanken, dass zumindest einige Kommas richtig gesetzt sind. Sie finden hier

,,

noch einige Kommas auf Vorrat, die Sie gegebenenfalls entsprechend Ihren Vorlieben im Text einsetzen können.

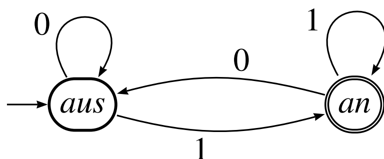
Kapitel 1

Endliche Automaten

Endliche Automaten sind einfache mathematische Modelle mit denen sich viele Arten von Hardware und Software beschreiben lassen. Endliche Automaten sind dadurch charakterisiert, dass sie sich immer in einem von endlich vielen „Zuständen" befinden und die Zustände durch „Eingaben" geändert werden können.

Beispiel 1.1. Wir betrachten einen Schalter als einfachen endlichen Automat.

- Zwei Zustände *an* und *aus* symbolisiert durch markierte Kreise.
- Zwei mögliche Eingaben 1 für anschalten und 0 für ausschalten.
- Initial ist der Schalter aus. Dies ist symbolisiert durch einen Pfeil aus dem Nichts.
- Das Ziel ist der eingeschaltete Zustand (der Zustand *an* soll erreicht werden) symbolisiert durch einen Doppelkreis.
- Der Schalter geht unabhängig vom Zustand auf *an* bei 1 und auf *aus* bei 0. Diese vier Zustandsübergänge sind durch markierte Pfeile symbolisiert.



Das Ziel wird erreicht durch jede Folge von Eingaben (0 oder 1), die mit einer 1 enden.

Bevor wir endliche Automaten formal definieren und dann genauer untersuchen, müssen wir zuerst noch einige wichtige Begriffe definieren.

Ein *Alphabet* ist eine endliche, nicht leere Menge von (Eingabe-)Zeichen, das meistens mit Σ bezeichnet wird. Im obigen Beispiel ist $\Sigma = \{0, 1\}$ die Menge, die aus den zwei möglichen Eingabezeichen 0 und 1 besteht. Ein anderes Alphabet wäre zum Beispiel die Menge der Kleinbuchstaben $\Sigma = \{a, b, c, \ldots, x, y, z\}$.

Ein *Wort* (oder *Zeichenkette* oder *String*) ist eine endliche Folge von Zeichen über einem bestimmten Alphabet. Zum Beispiel ist $w = 001101$ ein Wort über das Alphabet $\Sigma = \{0,1\}$. Die *Länge* eines Wortes ist die Anzahl der Zeichen in dem Wort. Die Länge eines Wortes w wird mit $|w|$ bezeichnet. Also gilt zum Beispiel $|001101| = 6$. Das leere Wort, das Wort der Länge 0, wird mit ε bezeichnet.

Potenzen eines Alphabets sind Mengen von Wörter eines Alphabets mit einer bestimmten Länge. Wir bezeichnen mit Σ^k die Menge aller Wörter über das Alphabet Σ der Länge k. Wenn $\Sigma = \{0,1\}$, dann ist zum Beispiel

$$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\} \ .$$

Die Menge aller Wörter über ein Alphabet beliebiger Länge (auch 0) wird mit Σ^* bezeichnet. Die Notation kommt daher, dass für * ein beliebiges k eingesetzt werden kann. Es gilt, dass

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \ldots = \bigcup_{k=0}^{\infty} \Sigma^k \ .$$

Wörter können *konkateniert* werden. Wenn x und y Worte sind, dann ist xy das konkatenierte Wort das mit dem Wort x beginnt und an das dann y angefügt wurde. Zum Beispiel ist das Wort $xy = 001101$ aus der Konkatenation der Wörter $x = 00$ und $y = 1101$ entstanden.

Es ist üblich als Zeichen von Alphabeten Zahlen (wie $0,1$) oder Kleinbuchstaben vom Anfang des Alphabets (wie $a,b,c,d,\ldots$) und für Namen von Wörtern Kleinbuchstaben vom Ende des Alphabets (wie $\ldots s,t,v,w,x,y,z$) zu verwenden.

Eine *Sprache* ist eine Untermenge von Wörtern über einem Alphabet. Für eine Sprache L gilt $L \subseteq \Sigma^*$ für ein Alphabet Σ. Der Begriff Sprache ist an die umgangssprachliche Bedeutung von „Sprache“ angelehnt. Alle korrekten deutschen Worte oder Sätze sind eine Untermenge von allen möglichen Zeichenketten über dem Alphabet aus Groß- und Kleinbuchstaben, den Umlauten und den Satzzeichen. Meist betrachtet man Sprachen mit speziellen Eigenschaften. Zum Beispiel sei L die Menge aller Zeichenketten über $\Sigma = \{0,1\}$, die mit einer 1 enden. Man schreibt auch formaler

$$L = \{x \in \Sigma^* \mid x \text{ endet mit einer } 1\}$$

oder

$$L = \{x1 \mid x \in \Sigma^*\} \ .$$

Für jedes beliebige Alphabet ist $\emptyset$ die leere Sprache. Die leere Sprache ist nicht zu verwechseln mit $\{\varepsilon\}$, der Sprache mit einem einzigen Wort, nämlich ε.

Ein *Problem* oder Wortproblem ist die Frage, ob ein Wort in einer Sprache enthalten ist oder nicht. Sei zum Beispiel L die Sprache aller Wörter, die mit 1 enden. Dann ist $001101 \in L$ aber $001110 \notin L$. Alle Berechnungen und Fragestellungen – umgangssprachlich „Problem“ – kann man als Wortproblem formulieren und damit darauf reduzieren festzustellen ob ein Wort $w \in \Sigma^*$ in einer Sprache $L \subseteq \Sigma^*$ ist.

1.1 Deterministische endliche Automaten

Definition 1.1. $A = (S, s_0, F, \Sigma, \delta)$ ist ein *deterministischer endlicher Automat (DEA)*, wenn die einzelnen Komponenten Folgendes bedeuten:

S Endliche Menge der möglichen Zustände des Automaten

s_0 Anfangszustand des Automaten, $s_0 \in S$

F Menge der Endzustände des Automaten, $F \subseteq S$

Σ Endliche Menge der Eingabezeichen, Alphabet

δ (Determinierte) Zustands-Überführungsfunktion, die gewissen Paaren (s, a) des kartesischen Produkts $S \times \Sigma$ einen Folgezustand s' aus S zuordnet. Man schreibt auch $\delta : S \times \Sigma \to S$ und $\delta(s, a) = s'$.

Falls *jedem* Paar (s, a) aus $S \times \Sigma$ ein Funktionswert zugeordnet ist, heißt die Überführungsfunktion *vollständig*, anderenfalls *partiell*. Eine partielle Überführungsfunktion kann durch die Einführung eines zusätzlichen Zustands zu einer vollständigen Überführungsfunktion ergänzt werden. Dazu werden in der Definition von δ alle bisher nicht definierten Paare (s, a) auf einen gemeinsamen neuen Zustand s_f, einen *Fehlerzustand*, abgebildet. Zusätzlich bilden wir alle Paare (s_f, a) auch auf s_f ab. Offensichtlich darf s_f kein Endzustand sein. Die Überführungsfunktion ist dann vollständig definiert.

Definition 1.2. $L(A)$ ist die von einem DEA A *akzeptierte Sprache*. Ein Automat A *akzeptiert* ein Wort $w = a_1a_2 \ldots a_n \in \Sigma^*$, wenn die Iteration $\delta(s_0, a_1) = s_{i_1}$, $\delta(s_{i_1}, a_2) = s_{i_2}, \ldots$ zu $\delta(s_{i_{n-1}}, a_n) = s_{i_n} \in F$ führt. Man schreibt dann auch $\delta(s_0, w) \in F$. Die Menge aller akzeptierten Worte bezeichnen wir als Sprache $L(A)$ des Automaten A. Es gilt also:

$$L(A) = \{w \in \Sigma^* \mid \delta(s_0, w) \in F\}$$

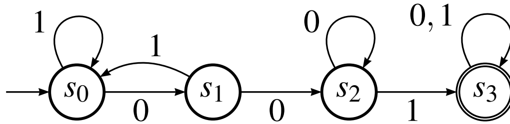
Falls das Wort $w = a_1a_2 \ldots a_n \in \Sigma^*$ nicht zu $L(A)$ gehört, bricht die Iteration $\delta(s_{i_{j-1}}, a_j) = s_{i_j}$ vorzeitig ab oder es gilt $s_{i_n} \notin F$. Beachten Sie, dass $\varepsilon \in L(A) \Leftrightarrow s_0 \in F$.

In Def. 1.2 haben wir die Überführungsfunktion δ mit Worten statt einzelnen Zeichen benutzt. Dies entspricht der sukzessiven Anwendung der Überführungsfunktion auf den jeweils neuen Zustand und das jeweils nächste Zeichen, bis das Wort vollständig abgearbeitet ist. Formal erweitern wir die Definition der Überführungsfunktion δ auf $S \times \Sigma^* \to S$ wie folgt:

- $\delta(s, \varepsilon) = s$ für alle Zustände $s \in S$
- $\delta(s, aw) = \delta(\delta(s, a), w)$ für alle Zustände $s \in S$, für alle Zeichen $a \in \Sigma$ und für alle Wörter $w \in \Sigma^*$.

Beispiel 1.2. Ein endlicher deterministischer Automat für die Sprache L, die alle Wörter mit Teilwort 001 enthält.

$$L = \{w \in \Sigma^* \mid w = x001y \text{ und } x, y \in \Sigma^*\}$$



$S = \{s_0, s_1, s_2, s_3\}$

s_0 ist Anfangszustand

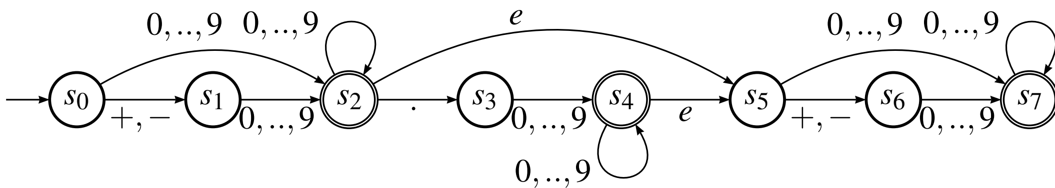
$F = \{s_3\}$

$\Sigma = \{0, 1\}$

δ Überführungsfunktion als Tabelle

δ	0	1
s_0	s_1	s_0
s_1	s_2	s_0
s_2	s_2	s_3
s_3	s_3	s_3

Beispiel 1.3. Endlicher deterministischer Automat für die normierte Darstellung reeller Zahlen.



$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$

s_0 ist Anfangszustand

$F = \{s_2, s_4, s_7\}$

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, ., e\}$

δ Überführungsfunktion als Tabelle

δ	$0,..,9$	$+,-$	.	e
s_0	s_2	s_1	$-$	$-$
s_1	s_2	$-$	$-$	$-$
s_2	s_2	$-$	s_3	s_5
s_3	s_4	$-$	$-$	$-$
s_4	s_4	$-$	$-$	s_5
s_5	s_7	s_6	$-$	$-$
s_6	s_7	$-$	$-$	$-$
s_7	s_7	$-$	$-$	$-$

Die Striche in der Tabelle besagen, dass δ für die entsprechende Kombination (s,a) nicht definiert ist. Um die partielle Überführungsfunktion δ zu vervollständigen, ersetzen wir die nicht definierten Überführungen $-$ durch einen Fehlerzustand s_f und überführen von s_f alle Zeichen auf s_f.

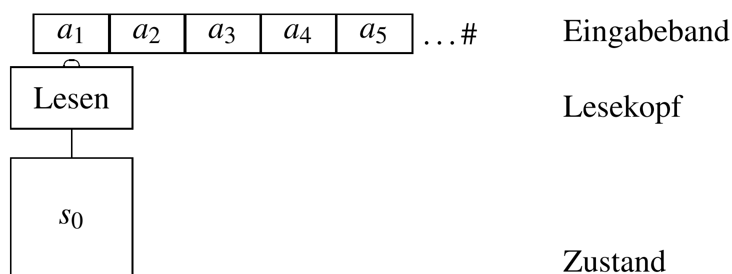
δ	$0,..,9$	$+,-$	$.$	e
s_0	s_2	s_1	s_f	s_f
s_1	s_2	s_f	s_f	s_f
s_2	s_2	s_f	s_3	s_5
s_3	s_4	s_f	s_f	s_f
s_4	s_4	s_f	s_f	s_5
s_5	s_7	s_6	s_f	s_f
s_6	s_7	s_f	s_f	s_f
s_7	s_7	s_f	s_f	s_f
s_f	s_f	s_f	s_f	s_f

1.2 Modell der Arbeitsweise eines Automaten

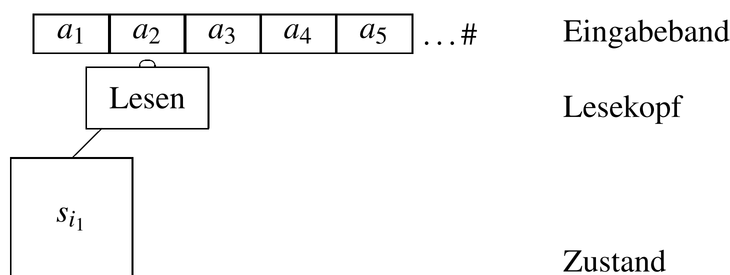
Man kann sich einen Automaten A als einen einfachen „Computer“ vorstellen, der aus einem Eingabeband, einem Lesekopf und einem inneren Zustand besteht.

Das Eingabeband kann nur zeichenweise von links nach rechts gelesen werden. Der „Computer“ ist „programmiert“ mit der Überführungsfunktion und kann die verschiedenen internen Zustände annehmen. Es ist *kein* Speicher für die Zeichen der Eingabe vorhanden.

Das Eingabeband ist von links mit den Zeichen eines Wortes $w = a_1a_2\dots a_n$ belegt. Danach folgt ein Bandzeichen #, das nicht zu dem Alphabet Σ gehört.

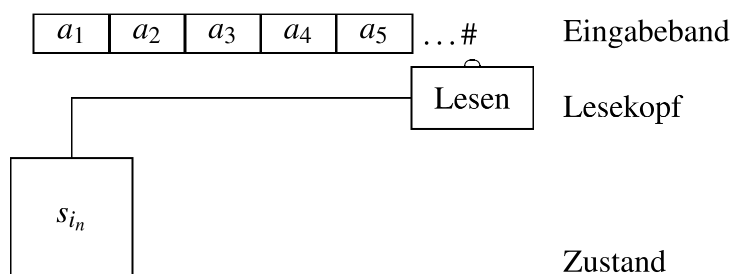


Initial ist der „Computer“ im Zustand s_0. Durch das Lesen des ersten Zeichens a_1 wandert der Lesekopf eine Position nach rechts. Der neue Zustand wird durch die Überführungsfunktion berechnet mit zum Beispiel $\delta(s_0,a_1) = s_{i_1}$. Es ergibt sich die folgende Konfiguration.



Der Lesekopf steht auf dem zweiten Zeichen und wird beim nächsten Durchgang a_2 lesen. Der innere Zustand ist s_{i_1}.

Dieser Prozess wird weiter fortgesetzt bis der „Computer" stehen bleibt, weil für das aktuelle Bandzeichen die Überführungsfunktion nicht definiert ist. Das ist spätestens dann der Fall, wenn der Lesekopf auf das spezielle Bandzeichen # trifft.



Falls der Lesekopf auf dem Bandzeichen # zum Stehen kommt und falls der letzte Zustand s_{i_n} ein Endzustand ist, dann gehört das eingelesene Wort zur Sprache des Automaten A. Es gilt also $w = a_1a_2a_3 \dots a_n \in L(A) \Leftrightarrow s_{i_n} \in F$.

1.3 Nicht-deterministische endliche Automaten

Definition 1.3. Das Tupel $A = (S, S_0, F, \Sigma, \delta)$ ist ein nicht-deterministischer endlicher Automat (NEA), wenn S, F und Σ die gleiche Bedeutung wie beim DEA haben und für die anderen Komponenten gilt:

S_0 Menge der Anfangszustände (von denen es also mehr als einen geben kann), $S_0 \subseteq S$

δ Überführungsfunktion, die gewissen Paaren (s, a) des kartesischen Produkts $S \times \Sigma$ mehrere mögliche Folgezustände, die man in einer Menge $T \subseteq S$ zusammenfassen kann, zuordnet. Man schreibt auch $\delta(s, a) = T$ und $\delta : S \times \Sigma \rightarrow P(S)$, wobei $P(S)$ die Potenzmenge von S bezeichnet.

Nicht definierte Übergänge entsprechen der Abbildung in die leere Menge. Man kann die Definition von δ erweitern, indem man auch Mengen von Zuständen als Ausgangspunkt zulässt. Für $T \subseteq S$ ist dann $\delta(T,a) = \bigcup_{t \in T} \delta(t,a)$. Die leere Menge dient dann als Fehlerzustand, mit der δ auf $S \times \Sigma$ eindeutig vervollständigt werden kann.

Genau wie beim deterministischen endlichen Automat können wir die Definition von δ auf $P(S) \times \Sigma^* \to P(S)$ erweitern, indem wir statt einzelner Zeichen wieder Worte zulassen.

- $\delta(T, \varepsilon) = T$ für alle Zustandsmengen $T \subseteq S$
- $\delta(T, aw) = \delta(\delta(T,a), w)$ für alle Zustandsmengen $T \subseteq S$, für alle Zeichen $a \in \Sigma$ und für alle Wörter $w \in \Sigma^*$.

Definition 1.4. $L(A)$ ist wie beim DEA die von einem NEA A *akzeptierte Sprache*. Wir betrachten ein beliebiges Wort $w = a_1 a_2 \dots a_n$ aus Σ^*. Ausgehend von S_0 erzeugen wir iterativ die Mengen

$$\delta(S_0, a_1) = T_1, \delta(T_1, a_2) = T_2, \dots, \delta(T_{n-1}, a_n) = T_n$$

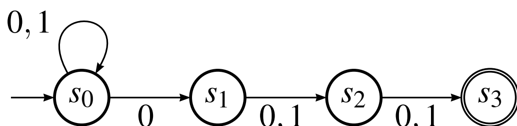
und erhalten mit T_n die Menge aller möglichen Zustände, die mit der Überführungsfunktion von den Anfangszuständen ausgehend durch Eingabe des Wortes w erreicht werden können.

Ist unter den Zuständen T_n ein Endzustand, dann soll das Wort w *akzeptiert* werden. In kürzerer Form können wir Folgendes schreiben:

$$L(A) = \{w \in \Sigma^* \mid \delta(S_0, w) \cap F \neq \emptyset\}$$

Es gilt, dass $\varepsilon \in L(A) \Leftrightarrow S_0 \cap F \neq \emptyset$.

Beispiel 1.4. Endlicher nicht-deterministischer Automat.



$S = \{s_0, s_1, s_2, s_3\}$, $S_0 = \{s_0\}$, $F = \{s_3\}$, $\Sigma = \{0,1\}$, die Überführungsfunktion δ als Tabelle

δ	0	1
s_0	$\{s_0, s_1\}$	s_0
s_1	s_2	s_2
s_2	s_3	s_3

Die Nicht-Determiniertheit besteht darin, dass vom Zustand s_0 zwei Pfeile mit dem Eingabezeichen 0 ausgehen. Das heißt es gibt zwei mögliche Zielzustände im Zustand s_0 beim Lesen des Zeichens 0; die Zustände s_0 und s_1. Welche Worte werden von dem NEA akzeptiert?

- Untersuchung für $w = 10101$ ergibt:

$$\begin{aligned}\delta(s_0,1) &= s_0,\\ \delta(s_0,0) &= \{s_0,s_1\},\\ \delta(\{s_0,s_1\},1) &= \{s_0,s_2\},\\ \delta(\{s_0,s_2\},0) &= \{s_0,s_1,s_3\},\\ \delta(\{s_0,s_1,s_3\},1) &= \{s_0,s_2\}\end{aligned}$$

  Da $\{s_0,s_2\} \cap F = \emptyset$ wird 10101 nicht akzeptiert.

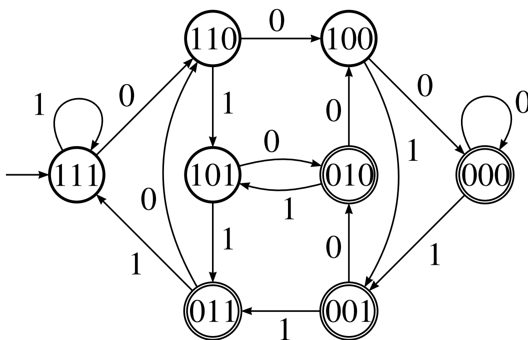
- Untersuchung für $w = 11001$ ergibt:

$$\begin{aligned}\delta(s_0,1) &= s_0,\\ \delta(s_0,1) &= s_0,\\ \delta(s_0,0) &= \{s_0,s_1\},\\ \delta(\{s_0,s_1\},0) &= \{s_0,s_1,s_2\},\\ \delta(\{s_0,s_1,s_2\},1) &= \{s_0,s_2,s_3\}\end{aligned}$$

  Da $\{s_0,s_2,s_3\} \cap F = \{s_3\} \neq \emptyset$ wird 11001 akzeptiert.

Im allgemeinen gilt, dass die Sprache des Automaten die Menge aller Bitfolgen, das heißt Worte über dem Alphabet $\Sigma = \{0,1\}$ ist, deren drittletzte Ziffer eine 0 ist.

Mit einiger Intuition kann man auch einen deterministischen Automaten finden, der dieselbe Sprache akzeptiert, aber wesentlich mehr Zustände besitzt.



Die Bezeichnungen der Zustände orientieren sich dabei an den möglichen Kombinationen der letzten drei aufeinander folgenden Bits.

Auch wenn nicht-deterministische Automaten auf den ersten Blick mächtiger erscheinen als deterministische, ist das nicht der Fall. Es gibt ein allgemeines Verfahren, mit dem zu jedem NEA ein DEA konstruiert werden kann, der die gleiche Sprache akzeptiert.

Satz 1.1. Zu jedem nicht-deterministischen endlichen Automaten gibt es einen deterministischen endlichen Automaten, der die gleiche Sprache akzeptiert.

Beweis. Sei $A = (S, S_0, F, \Sigma, \delta)$ nicht-deterministisch. Wir definieren einen neuen Automaten $A' = (S', s'_0, F', \Sigma', \delta')$ aus den Komponenten von A wie folgt:

1. $S' = \{t_T \mid T \subseteq S$ bzw. $T \in P(S)\}$, d.h. jede Untermenge $T \subseteq S$ entspricht einem Zustand t_T des Automaten A'.
2. $s'_0 = t_{S_0}$, der neue Anfangszustand entspricht dem Zustand, der alle ursprünglich Anfangszustände enthält.
3. $F' = \{t_T | T \cap F \neq \emptyset\}$, die neuen Endzustände sind alle Zustände, die mindestens einen ursprünglichen Endzustand enthalten.
4. $\Sigma' = \Sigma$, das gleiche Alphabet.
5. $\delta'(t_T, a) = t_{T'}$ mit $T' = \delta(T, a)$, T' beinhaltet alle möglichen ursprünglichen Zielzustände eines jeden ursprünglichen Zustands in T.

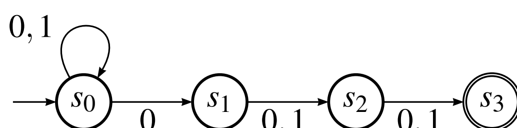
Der Automat A' ist ein deterministischer endlicher Automat per Konstruktion. Sei nun $w = a_1 a_2 \ldots a_n \in \Sigma^*$, dann gelten folgende Äquivalenzaussagen:

$$\begin{aligned} w \in L(A) &\Leftrightarrow \delta(S_0, a_1) = T_1, \delta(T_1, a_2) = T_2, \ldots, \delta(T_{n-1}, a_n) = T_n \text{ mit } T_n \cap F \neq \emptyset \\ &\Leftrightarrow \delta'(t_{S_0}, a_1) = t_{T_1}, \delta'(t_{T_1}, a_2) = t_{T_2}, \ldots, \delta'(t_{T_{n-1}}, a_n) = t_{T_n} \text{ mit } t_{T_n} \in F' \\ &\Leftrightarrow w \in L(A') \end{aligned}$$

Bei der Konstruktion des deterministischen Automaten muss unter Umständen mit großen Zustandsmengen gerechnet werden, da eine Menge mit m Elementen 2^m Untermengen besitzt. □

Mit der *Teilmengen-Konstruktion* konstruieren wir einen deterministischen Automaten wie in Satz 1.1 angegeben aus einem nicht-deterministischen. Allerdings werden nur die Teilmengen T (beziehungsweise Zustände t_T) erzeugt, die von einem Anfangszustand aus erreicht werden können. Dazu gehen wir von der Menge der initialen Zustände aus und konstruieren dann schrittweise für jedes Eingabezeichen die Menge von Zielzuständen, die von den Zuständen der jeweiligen Ausgangsteilmengen mit der Überführungsfunktion erreicht werden können.

Beispiel 1.5. Wir konstruieren den deterministischen aus dem nicht-deterministischen Automaten von Beispiel 1.4 und geben Graphen und Überführungsfunktion δ an.



δ	0	1
s_0	$\{s_0, s_1\}$	s_0
s_1	s_2	s_2
s_2	s_3	s_3
$\underline{s_3}$	$\emptyset$	$\emptyset$

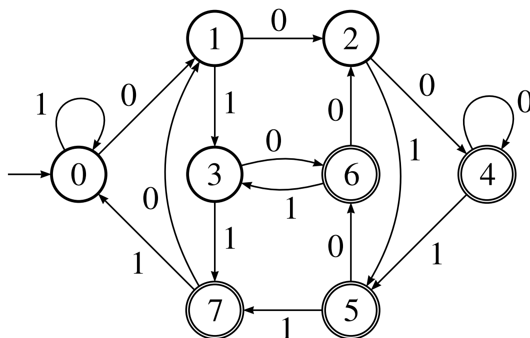
Wir konstruieren nun die Überführungsfunktion δ' des DEA, dessen Zustände aus Zustandsmengen des NEA bestehen. Die Menge von Anfangszuständen ist $\{s_0\}$, die Menge von Endzuständen ist $\{s_3\}$. Im ersten Schritt können wir von s_0 mit Eingabe 0 die Zustände s_0 und s_1 erreichen und mit Eingabe 1 nur den Zustand s_0 erreichen. Beachten Sie, dass wir in der Tabelle bei Mengen mit einem Zustand die geschweiften Klammern auch weglassen, um und Schreibarbeit zu sparen. Alle Zustandsmengen, die wir erreichen können, betrachten wir wieder als Menge von Zuständen für die wir δ' berechnen müssen. Wenn eine Zustandsmenge schon einmal generiert wurde, müssen wir diese nicht noch einmal betrachten. Dadurch generieren wir nur erreichbare Zustandsmengen. Daraus können wir dann direkt die Überführungsfunktion δ' als Tabelle angeben.

	0	1
$\{s_0\}$	$\{s_0, s_1\}$	$\{s_0\}$
$\{s_0, s_1\}$	$\{s_0, s_1, s_2\}$	$\{s_0, s_2\}$
$\{s_0, s_1, s_2\}$	$\{s_0, s_1, s_2, s_3\}$	$\{s_0, s_2, s_3\}$
$\{s_0, s_2\}$	$\{s_0, s_1, s_3\}$	$\{s_0, s_3\}$
$\underline{\{s_0, s_1, s_2, s_3\}}$	$\{s_0, s_1, s_2, s_3\}$	$\{s_0, s_2, s_3\}$
$\underline{\{s_0, s_2, s_3\}}$	$\{s_0, s_1, s_3\}$	$\{s_0, s_3\}$
$\underline{\{s_0, s_1, s_3\}}$	$\{s_0, s_1, s_2\}$	$\{s_0, s_2\}$
$\underline{\{s_0, s_3\}}$	$\{s_0, s_1\}$	$\{s_0\}$

δ'	0	1
z_0	z_1	z_0
z_1	z_2	z_3
z_2	z_4	z_5
z_3	z_6	z_7
$\underline{z_4}$	z_4	z_5
$\underline{z_5}$	z_6	z_7
$\underline{z_6}$	z_2	z_3
$\underline{z_7}$	z_1	z_0

Wir geben dabei jedem Zustand einen anderen Namen. Im Beispiel nummerieren wir mit i die erreichbaren Zustandsmengen durch und vergeben den Namen z_i, der Startzustand ist z_0. Endzustände werden alle Zustände, deren Menge von Zuständen mindestens einen Endzustand enthält, F' ist $\{z_4, z_5, z_6, z_7\}$.

Abschließend sehen wir die graphische Darstellung des DEA, der bis auf die Bezeichnung der Zustände identisch ist mit dem zweiten Graphen aus Beispiel 1.4 auf Seite 12.

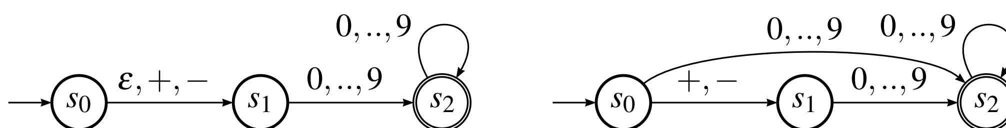


1.4 Endliche Automaten mit ε-Übergängen

Wir haben bisher für das leere Wort ε und jeden Zustand s eines endlichen nicht-deterministischen Automaten $\delta(s,\varepsilon) = \{s\}$ angenommen. Eine Erweiterung besteht darin, dass wir Übergänge zu anderen Zuständen für die leere Zeichenkette ε zulassen. Für die Überführungsfunktion der Zustände gilt dann $\delta(s,\varepsilon) = T \supseteq \{s\}$. Wir nennen die Automaten mit spontanen Übergängen beziehungsweise mit ε-Übergängen dann εNEA.

Endliche Automaten mit ε-Übergängen erlauben eine kompaktere Beschreibung von einigen Sprachen.

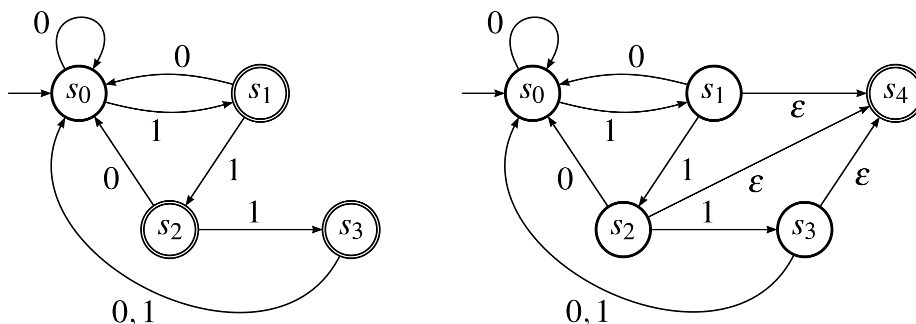
Beispiel 1.6. Endlicher Automat *mit* ε-Übergängen und *ohne* ε-Übergänge für ganze Zahlen mit optionalem Vorzeichen.



Wir wir sehen werden, sind endliche Automaten mit ε-Übergängen allerdings nicht mächtiger als endliche Automaten ohne ε-Übergänge.

Hilfssatz 1.2. In einem endlichen Automaten können durch die Einführung von ε-Übergängen ein eindeutiger Anfangszustand und ein eindeutiger Endzustand hergestellt werden.

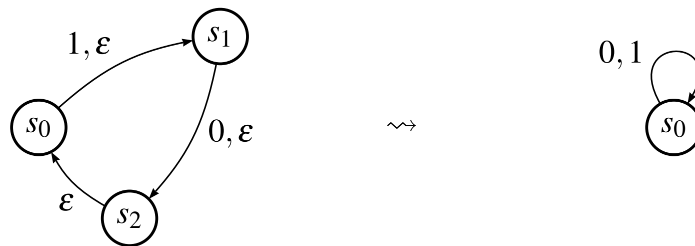
Wir zeigen das an einem Beispiel.



Satz 1.3. Zu jedem endlichen Automaten mit ε-Übergängen gibt es einen ohne ε-Übergänge, der die gleiche Sprache akzeptiert.

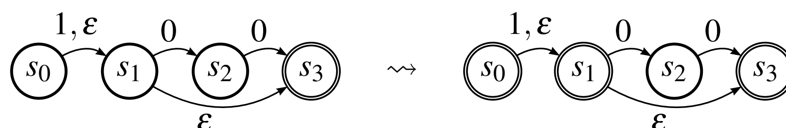
Der folgende Algorithmus konstruiert graphisch aus einem endlichen Automaten mit ε-Übergängen einen äquivalenten ohne ε-Übergänge:

1. Ein ε-Zyklus ist eine Menge zwei oder mehr Zuständen, bei denen jeder Zustand alle anderen durch ε-Übergänge erreicht. Falls ε-Zyklen existieren, fasse alle Zustände eines Zyklus zu einem zusammen und übernehme alle von ε verschiedenen Eingabezeichen des Zyklus in einer Schleife. Beispiel:

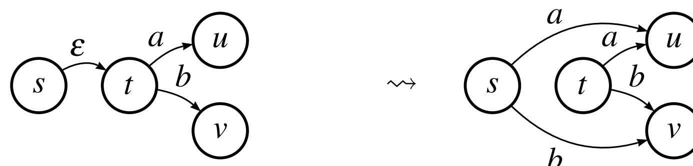


Falls ein Zustand in einem ε-Zyklus ein Endzustand ist, dann ist der zusammengefasste Zustand ein Endzustand.

2. Mache jeden Zustand s, von dem aus eine ε-Übergangssequenz in einen Endzustand führt, selbst zu einem Endzustand. Beispiel:



3. Entferne bei jeder Übergangsfolge, die von s mit ε zu t und dann mit a zu u führt, den ε-Übergang und ersetze ihn durch einen Übergang von s mit a nach u.



Übertragen Sie für diesen Schritt zunächst alle Zustände in ein neues Diagramm, damit Sie nicht ε-Übergange eliminieren, die noch gebraucht werden. Beachten Sie, dass alle restlichen Übergänge, die von s mit $a \neq \varepsilon$ nach t übergehen unverändert bleiben. Eliminieren Sie die ε schrittweise, wählen Sie zunächst die ε, die am Ende eine ε-Übergangssequenz stehen.

4. Entferne alle im neuen Diagramm nicht mehr erreichbaren Zustände.

Es gibt also für jeden nicht-deterministischen endlichen Automaten mit ε-Übergängen einen deterministischen endlichen Automaten, der die gleiche Sprache akzeptiert. Wir können diesen DEA konstruieren indem wir erst mit der ε-Elimination aus dem εNEA einen NEA erzeugen und dann mit der Teilmengenkonstruktion einen DEA. Alle diese Automatentypen, vom komplexen εNEA bis zum einfachen DEA, sind also gleich mächtig. Wir untersuchen jetzt, ob auch ein DEA noch weiter vereinfacht werden kann.

1.5 Äquivalenz und Minimierung von Automaten

Zwei Automaten heißen *äquivalent*, wenn sie die gleiche Sprache akzeptieren. Äquivalente Automaten können unterschiedliche Zustände und andere Überführungsfunktionen haben. Wir werden jedoch sehen, dass es für jede Sprache, also auch für jeden endlichen Automaten, einen bis auf Bezeichnungen eindeutigen deterministischen endlichen Automaten gibt, den sogenannten *Minimalautomaten*. Wir gehen im Folgenden von deterministischen Automaten aus. Nicht-deterministische Automaten werden gegebenenfalls erst in einen äquivalenten deterministischen Automat umgewandelt. Die Reduktion auf einen Minimalautomaten erfolgt durch das Zusammenfassen sogenannter *äquivalenter* Zustände.

Definition 1.5. Zwei Zustände s und s' eines endlichen deterministischen Automaten heißen *äquivalent* ($s \cong s'$), wenn die Menge der Worte, die in einen Endzustand führen, für beide identisch ist, das heißt, wenn $\delta(s,w) \in F \Leftrightarrow \delta(s',w) \in F$ für alle $w \in \Sigma^*$ gilt.

Wenn zwei Zustände nicht äquivalent sind, dann heißen die beiden Zustände *unterscheidbar*. Wenn zwei Zustände unterscheidbar sind, dann gibt es ein Wort, das ausgehend von dem einem Zustand in einen Endzustand, aber von dem anderen Zustand in einen Nicht-Endzustand überführt wird.

Ein Automat heißt *reduziert*, beziehungsweise *minimal*, wenn keine äquivalenten Zustände existieren und jeder Zustand von s_0 aus erreichbar ist. Alle zueinander äquivalenten Zustände können in einer *Äquivalenzklasse* $[s]$, die durch einen Vertreter s repräsentiert werden kann, zusammengefasst werden.

Satz 1.4. Zu jedem deterministischen endlichen Automaten gibt es einen reduzierten, der die gleiche Sprache akzeptiert.

Beweis. Der Beweis erfolgt durch Angabe eines Verfahrens zur schrittweisen Bildung der Äquivalenzklassen $[s]$ des Automaten $A = (S, s_0, F, \Sigma, \delta)$ und der Definition des *reduzierten Automaten* $A' = (S', s'_0, F', \Sigma, \delta')$ mit Hilfe der gewonnen Äquivalenzklassen:

$S' = \{[s] | s \in S\}$, ein Vertreter jeder Äquivalenzklasse

$s'_0 = [s_0]$, Äquivalenzklasse, die s_0 enthält

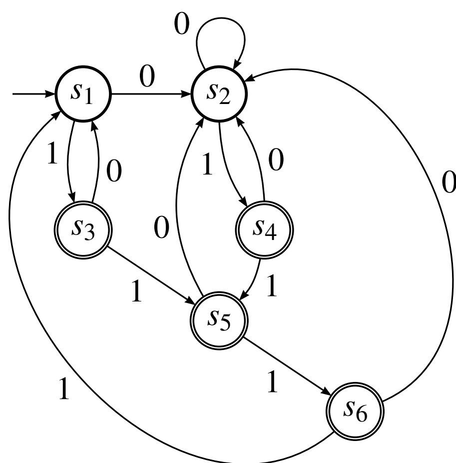
$F' = \{[s] | s \in F\}$, ein Vertreter jeder Äquivalenzklasse der Endzustände

$\delta'([s],a) = [\delta(s,a)]$ für alle $[s] \in S'$ und $a \in \Sigma$, nur noch Überführungen von einem Vertreter jeder Äquivalenz

□

Um einen reduzierten Automaten zu erhalten, identifizieren wir alle paarweise unterscheidbaren Zustände. Die übrig gebliebenen Paare von Zuständen sind dann in der gleichen Äquivalenzklasse. Wir reduzieren zunächst in einem Beispiel intuitiv einen Automaten und geben dann ein vollständiges Verfahren an.

Beispiel 1.7. Wir betrachten folgende Überführungsfunktion und graphische Darstellung eines deterministischen endlichen Automaten.

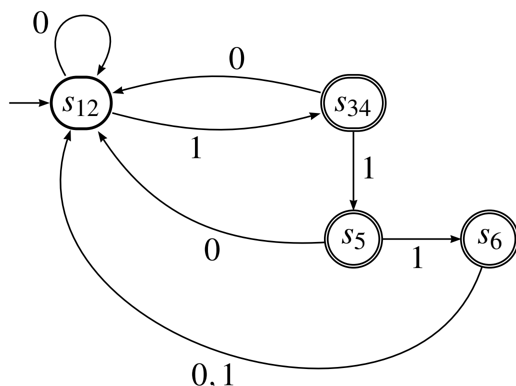


δ	0	1
s_1	s_2	s_3
s_2	s_2	s_4
s_3	s_1	s_5
s_4	s_2	s_5
s_5	s_2	s_6
s_6	s_2	s_1

Wir wissen von vornherein, dass Endzustände von Nicht-Endzuständen unterscheidbar sind. Es sind also alle Paare in $\{s_1, s_2\} \times \{s_3, s_4, s_5, s_6\}$ unterscheidbar. Durch Probieren stellen wir weiter fest, dass die Zustände s_5 und s_6 unterscheidbar sind unter der Eingabe 1, da $\delta(s_5, 1) = s_6 \in F$ und $\delta(s_6, 1) = s_1 \notin F$. Ähnlich geht es nun weiter mit s_4 und s_5, da $\delta(s_4, 1) = s_5$ und $\delta(s_5, 1) = s_6$. Da s_5 und s_6 unterscheidbar sind, muss auch s_4 von s_5 unterscheidbar sein. Wir können dies auch direkt sehen, indem wir die zur Unterscheidung benutzten Worte zusammensetzen; $\delta(s_4, 11) = s_6 \in F$ und $\delta(s_5, 11) = s_1 \notin F$. Genauso finden wir, dass s_3 von s_5 und von s_6 unterscheidbar ist. Weiteres Probieren führt zu keinen unterscheidbaren Zuständen mehr. Wir erhalten also folgende Äquivalenzklassen:

$$\{s_1, s_2\}, \{s_3, s_4\}, \{s_5\}, \{s_6\}$$

Der reduzierte Automat ist dann



Der folgende Algorithmus bestimmt die Äquivalenzklassen durch Markierung aller Zustandspaare, die nicht äquivalent sein können.

```
1  U = ∅
2  for (s,t) ∈ ((S − F) × F) ∪ (F × (S − F)):
3    U = U ∪ {(s,t)}
4  changed = True
5  while changed:
6    changed = False
7    for (s,t) ∈ ((S × S) − U and s ≠ t:
8      if ∃a ∈ Σ : (δ(s,a), δ(t,a)) ∈ U:
9        U = U ∪ {(s,t)}
10       changed = True
```

Wir konstruieren die Menge U der unterscheidbaren Zustände. Initial sind alle Paare aus Endzuständen und Nicht-Endzuständen unterscheidbar. Wir versuchen dann neue Paare von unterscheidbaren Zuständen zu generieren, indem wir jedes Paar (s,t) von bisher nicht unterscheidbaren Zuständen untersuchen. Wenn es ein Zeichen gibt, das von s und t in zwei unterscheidbare Zustände führt, dann sind auch s und t unterscheidbar. Wir wiederholen dies solange, bis bei einem kompletten Durchlauf keine neuen unterscheidbaren Zustände dazu kommen. Der Algorithmus kann mit folgendem Schema illustriert werden:

	s_1	s_2	s_3	s_4	s_5	s_6
s_1						
s_2						
s_3	$\times_0$	$\times_0$				
s_4	$\times_0$	$\times_0$				
s_5	$\times_0$	$\times_0$	$\times_2$	$\times_2$		
s_6	$\times_0$	$\times_0$	$\times_1$	$\times_1$	$\times_1$	

Wir berücksichtigen nur die Paare (s_i, s_j) für die $i > j$ gilt, da wenn s_i von s_j unterscheidbar ist auch s_j von s_i unterscheidbar ist. Die Markierungen $\times_i$ identifizieren die Paare von Zuständen, die unterscheidbar sind. Es sind $\times_0$ die Anfangsmarkierungen, die sich durch die Unterscheidung der Endzustände und Nicht-Endzustände ergeben. Die Markierungen $\times_1$ und $\times_2$ sind die Zustandspaare, die beim ersten beziehungsweise zweiten Durchlauf als nicht äquivalent erkannt werden. Die nicht markierten Zustandspaare, die Paare (s,t), die nicht in U sind, stellen die Äquivalenzklassen dar.

$$
\begin{array}{rcll}
[s_1] = [s_2] & = & \{s_1, s_2\} & \hat{=} \; s_{12} \\
[s_3] = [s_4] & = & \{s_3, s_4\} & \hat{=} \; s_{34} \\
[s_5] & & & \hat{=} \; s_5 \\
[s_6] & & & \hat{=} \; s_6
\end{array}
$$

Satz 1.5. Die Reduktion zweier Automaten, welche die gleiche Sprache akzeptieren, führt, abgesehen von der Benennung der Zustände, auf den gleichen reduzierten Automaten.

1.6 Anwendung in der Texterkennung

Mit Erkennern für reguläre Sprachen können wir eine lexikalische Analyse durchführen und so Patterns in Texten extrahieren.

1.6.1 Ein einfaches Pattern-Matching Problem

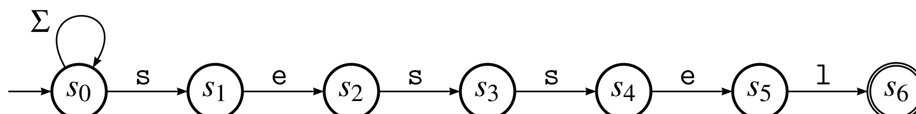
Gegeben sei ein Wort $x = x_1 \dots x_n$ ($|x| = n$), und ein kürzeres Wort (Pattern) $p = p_1 \dots p_m$. Es soll jedes Vorkommen des *Patterns* (*Musters*) p im Wort x festgestellt werden. Ein einfacher Algorithmus, der ohne Automatenmodell auskommt, vergleicht ab allen möglichen Startstellen in x jeweils alle Zeichen in p bis entweder eine Stelle ungleiche Zeichen hat oder alle Stellen abgearbeitet sind und damit eine Fundstelle identifiziert wurde.

```
1   def suche_ab(p, x, s):
2     for i ∈ {1,...,|p|}:
3       if  p_i ≠ x_{s+i}:
4         return False
5     return True
6
7   def suche(p, x):
8     for s ∈ {1,...,|x| − |p|}:
9       if  sucheab(p, x, s):
10        print "gefunden an", s
11        # weitere Verarbeitung Fundstelle
```

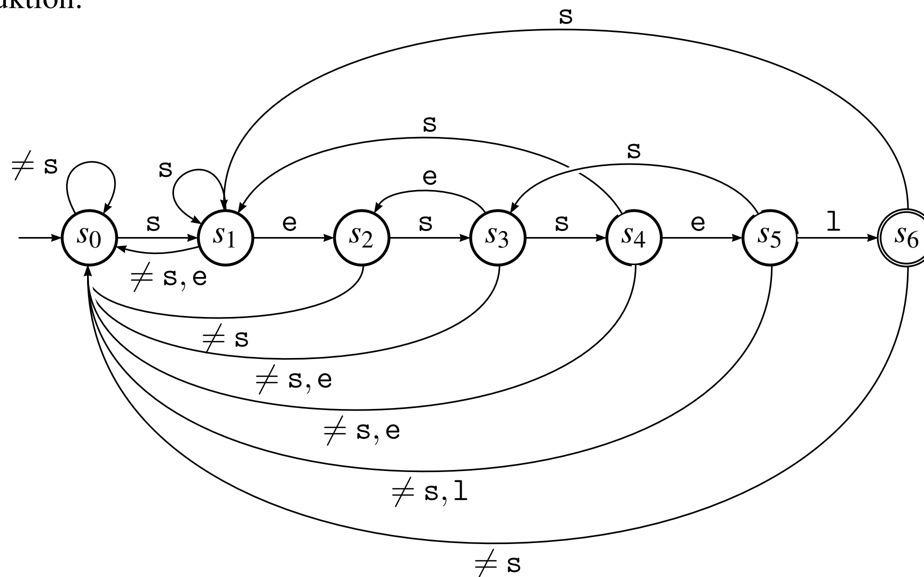
Es werden alle Indizes der Fundstellen von p in x ausgegeben. Die Anzahl der durchzuführenden Vergleiche ist abhängig von der Größenordnung $(n-m) \cdot q$ mit $1 < q < m$. Dabei ist q die von der Länge m und der Art des Worts p abhängige durchschnittliche Anzahl der Durchläufe der Schleife in suche_ab. Im schlechtesten Fall ist dies quadratisch zu $n = |s|$, wenn zum Beispiel $x = aaaa \dots aaaa$ und $p = aa \dots aa$ mit $|p| = |x|/2$.

Mit Hilfe des Automatenmodells kann man Algorithmen konstruieren, bei denen die Anzahl der durchzuführenden Vergleiche nur von n abhängig ist. Dabei wird die Struktur des Musters p betrachtet und ausgenutzt, wenn sich Buchstabenfolgen am Anfang des Wortes auch innerhalb des Wortes noch einmal wiederholen.

Wir betrachten dazu als Beispiel das Pattern `sessel`, dessen Vorkommen in einem Text zu suchen sei. Ein (nicht-deterministischer) Automat, der sich zum Auffinden des Patterns an einer beliebigen Stelle in einem Text eignet, ist:



Dabei steht Σ für das gesamte Alphabet. Wegen der Nicht-Determiniertheit ist der Automat nicht unmittelbar für die Erstellung eines Algorithmus nutzbar. Deswegen bestimmt man den zugehörigen deterministischen Automaten, zum Beispiel mit Hilfe der Teilmengen-Konstruktion:

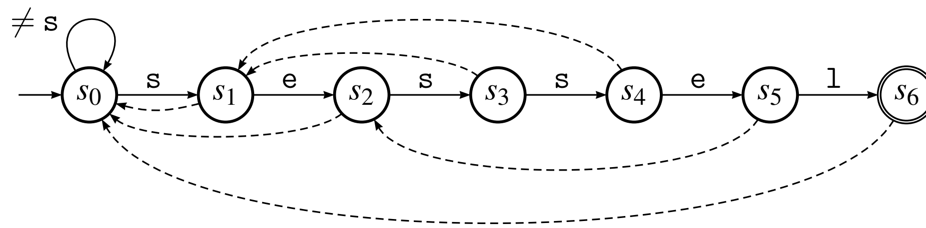


Mit der Überführungsfunktion δ können wir nun leicht einen Algorithmus für die Mustererkennung formulieren. Wir starten mit Zustand s_0 und lesen das Wort x Zeichen für Zeichen, wobei jedes Mal der Folgezustand mit Hilfe von δ neu bestimmt wird. Immer dann, wenn ein *Erkennungszustand* wie s_6 auftritt, erfolgt die Verarbeitung der Fundstelle oder eine Nachricht. Für den allgemeinen Fall erhalten wir folgenden Code.

```
1  def suche(p, x):
2      t = s_0
3      for i ∈ {1,...,|x|}:
4          t = δ(t, x_i)
5          if t ∈ F:
6              print "found at ", i −|p|+1
7              # weitere Verarbeitung Fundstelle
```

Der Algorithmus benötigt n Schritte, um alle Vorkommen des Musters p in einem Text x zu erkennen und ist insofern optimal. Allerdings ist die Bestimmung der Überführungsfunktion δ nicht bei dem Rechenaufwand berücksichtigt.

Es gibt noch einen anderen Weg, bei dem keine deterministische Überführungsfunktion zu einem NEA konstruiert werden muss. Stattdessen wird der sogenannte Skelettautomat und eine sogenannte failure-Funktion benutzt. Dies kann bei dem vorliegenden Beispiel mit folgender Grafik beschrieben werden. Die gestrichelten Pfeile stellen die failure-Funktion dar. Der Rest ist der Skelettautomat, der eine deterministische, partielle Überführungsfunktion besitzt.



Die failure-Funktion gibt an, in welchen Zustand man (spontan) zurückzugehen hat, wenn für das aktuelle einzulesende Zeichen x_i die Überführungsfunktion des Skelettautomaten nicht definiert ist. Dieser Zustand ist so gewählt, dass ein möglichst großes Anfangsstück von p mit dem Ende des bis dahin eingelesenen x-Textes (außer x_i!) übereinstimmt. Passt auch da x_i nicht, wählt man mit der failure-Funktion das Anfangsstück von p kürzer. Spätestens im Anfangszustand endet das Zurückgehen, weil dort die Überführungsfunktion für alle Zeichen definiert ist. Mit Hilfe der Überführungsfunktion δ und der failure-Funktion lautet der Algorithmus:

```
1 def suche(p, x):
2     t = s_0
3     for i ∈ {1,...,|x|}:
4         while δ(t,x_i) == FAIL:
5             t =failure(t)
6         t = δ(t,x_i)
7         if t ∈ F:
8             print "found at ", i −|p|+1
9             # weitere Verarbeitung Fundstelle
```

Wir bestimmen nun die failure-Funktion. Es seien $s_0,\dots,s_m$ die Zustände des Skelettautomaten, wobei $|p| = m$ die Länge des Musters p ist. Der Zustand s_0 hat keinen failure-Wert; für s_1 gilt failure$(s_1) = s_0$. Wir nehmen an, dass die Werte der failure-Funktion für die Zustände s_i mit $i = 1,2,\dots,j-1$ schon berechnet sind und berechnen failure(s_j). Dazu betrachten wir das Zeichen y_j des Wortes $y = y_1 \dots y_{j-1}y_j \dots y_m$, für das gilt, dass $\delta(s_{j-1}, y_j) = s_j$. Von $t = s_{j-1}$ ausgehend bestimmen wir mit $t =$ failure(t) schrittweise den vorausgehenden Wert der failure-Funktion und überprüfen, ob $\delta(t, y_j)$ existiert. Sobald dies der Fall ist, haben wir in $\delta(t, y_j)$ den Wert von failure(s_j) gefunden. Wir repräsentieren die Fehlerfunktion failure durch ein Feld. Es ergibt sich:

```
1  failure[s_1] = s_0
2  for j ∈ {2,3,...,|p|}:
3      t = failure[s_{j−1}]
4      while δ(t,y_j)== FAIL: # nicht definiert
```

```
5         t = failure[t]
6     failure[s_j] = δ(t, y_j)
```

Der Rechenaufwand für die Bestimmung der failure-Funktion (die Tabelle) hat die Größenordnung m. Die **while**-Schleife wird jeweils maximal zweimal durchlaufen. Beachten Sie, dass von dem Zustand s_0 ausgehend die failure-Funktion immer definiert ist ($\forall a \in \Sigma : \delta(s_0, a) \neq$FAIL). Für obiges Beispiel erhalten wir

```
1 failure[s_1] = s_0
2 failure[s_2] = s_0
3 failure[s_3] = s_1
4 failure[s_4] = s_1
5 failure[s_5] = s_2
6 failure[s_6] = s_0
```

Der Algorithmus wurde zuerst von Knuth, Morris und Pratt (KMP-Algorithmus) entdeckt und hat eine lineare Laufzeit abhängig von n + m im schlechtesten Fall[1]. Dies ist eine signifikante Verbesserung im Vergleich zum naiven quadratischen Algorithmus.

1.6.2 Gleichzeitiges Suchen nach mehreren Wörtern

Wir übertragen den Algorithmus, den wir zuvor für ein Wort ausgeführt haben, auf die Suche von mehreren Worten (Mustern) in einem Text. Die Aufgabe ist dabei, die failure-Funktion zu bestimmen.

Der Skelettautomat wird zu einem „Skelettbaum", bei dem gemeinsame Präfixe einen gemeinsamen Strang bilden. Bei unterschiedlichen Zeichen spaltet sich der Strang auf. Wir bezeichnen mit m die Zahl der Zeichen des längsten aufzufindenden Wortes. Dies entspricht der *Höhe* des Skelettautomaten. Der Algorithmus startet, indem failure$[s] = s_0$ gesetzt wird für alle Zustände mit dem Abstand $k = 1$ zur Wurzel, also der *Tiefe* $k = 1$. Danach werden die Werte der failure-Funktion für alle Zustände mit größerer Tiefe wiederum mit Hilfe der δ-Funktion und den schon bekannten Werten der failure-Funktion berechnet.

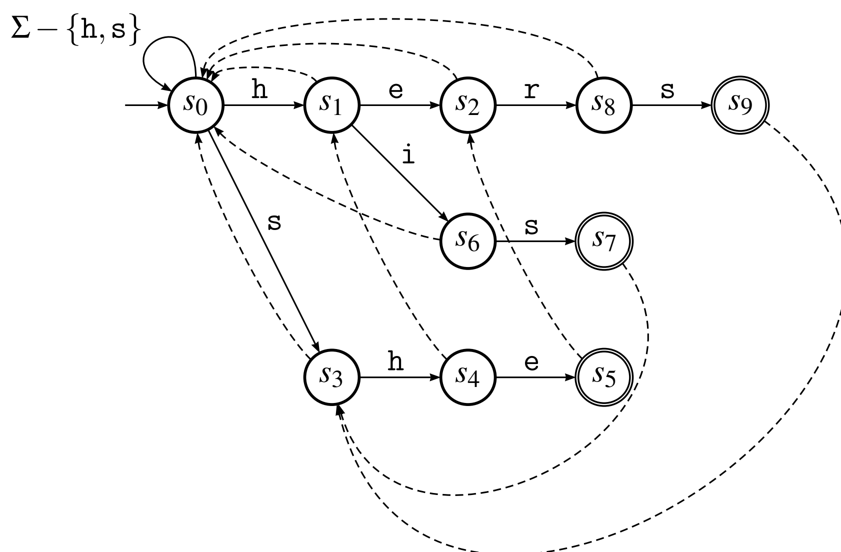
```
1 for s ∈ {δ(s_0, y_1) | y_1 erstes Zeichen eines Suchworts}: # Tiefe 1
2     failure[s] = s_0
3 for k ∈ {2, 3, ..., m}: # fuer alle Tiefen
4     for s ∈ {δ(t, y_k) | y_k das k.te Zeichen eines Suchworts}: # Tiefe k
5         Sei t, y_k so, dass s = δ(t, y_k)
6         t = failure[t]
7         while δ(t, y_k) ≠ FAIL: # nicht definiert
```

[1] Falls Ihnen die verschachtelte Schleife in suche Probleme macht, dann machen Sie sich klar, dass man im kompletten Durchlauf nicht häufiger zurückgehen kann als man Zeichen eingelesen hat. Also kann auch $t =$ failure$[t]$ maximal n mal aufgerufen werden.

```
8        t = failure[t]
9    failure[s] = δ(t, y_k)
```

Beispiel 1.8. Jedes Auftauchen eines der Worte { `hers`, `his`, `she` } soll in einem Text festgestellt werden. Der Skelettautomat hat eine Tiefe von $m = 4$.

Wir bleiben im Zustand s_0 für alle Zeichen außer den ersten Zeichen, also `h` oder `s`. Bei Tiefe 1 bildet die failure-Funktion die Zustände s_1 und s_3 auf s_0 ab.



Bei Tiefe 2 betrachten wir zunächst s_2. Wir gelangen zu s_2 mit $\delta(s_1, \texttt{e})$. Es ist also $s = s_2$, $t = s_1$ und $y_2 = \texttt{e}$. Wir springen mit der failure-Funktion zurück auf s_0. Dort ist $\delta(s_0, \texttt{e}) = s_0$ definiert. Also ist $\mathsf{failure}[s_2] = s_0$.

Betrachten wir nun s_4. Wir gelangen zu s_4 mit $\delta(s_3, \texttt{h})$. Es ist also $s = s_4$, $t = s_3$ und $y_2 = \texttt{h}$. Wir springen mit der failure-Funktion zurück auf s_0. Dort ist $\delta(s_0, \texttt{h}) = s_1$ definiert. Also ist $\mathsf{failure}[s_4] = s_1$. Die anderen Werte der failure-Funktion werden genauso berechnet.

1.6.3 Lexikalische Analyse

Das Hauptproblem der lexikalischen Analyse eines Quellprogramms in einer Programmiersprache besteht darin, aus dem Text die sogenannten *Tokens* – das sind die *Schlüsselworte* wie zum Beispiel for, if, while, not, and, or..., die *Operatoren* wie zum Beispiel +, -, * ,..., die *Zahlen- und Textkonstanten* und die *Variablennamen* herauszufinden.

Die bisherigen Beispiele und Anwendungen endlicher Automaten können zur Lösung dieses Problems herangezogen werden. Bei der Zusammensetzung und Implementierung dieser Bestandteile sind zusätzliche Dinge zu beachten. Dazu gehört zum Beispiel, dass zwischen Tokens teilweise Delimiter vorhanden sind, teilweise aber auch nicht. Das führt

dazu, dass ein beim Einlesen erkanntes Token beim weiterem Einlesen sich als Teil eines größeren Tokens herausstellen kann. Hier ist eine Strategie des „längsten passenden Stücks“ zu verfolgen. Dieses und ähnliche Probleme der lexikalischen Analyse werden im Rahmen dieser Vorlesung nicht weiter behandelt und sind klassische Themen des Compilerbaus.

Kapitel 2

Reguläre Sprachen

Reguläre Ausdrücke sind eine Notation zur Darstellung von Sprachen, die häufig in der Praxis verwendet wird. Man findet sie entweder als Standard-Bibliotheken oder als Sprachbestandteil in vielen Programmiersprachen. Spezielle Tools zur Syntaxanalyse (z.B. `lex`) oder Kommandozeilentools (z.B. `egrep`) sind rund um reguläre Ausdrücke aufgebaut. Reguläre Ausdrücke sind eng verwandt mit nicht-deterministischen endlichen Automaten und können als alternative Darstellungsform betrachtet werden.

2.1 Reguläre Ausdrücke und Sprachen

Ein *regulärer Ausdruck* besteht aus Zeichen eines Alphabets oder anderen regulären Ausdrücken, die durch die Operationen *Verkettung*, *Iteration (*)* oder *Wahlmöglichkeit* ($|$) miteinander verbunden sind. Jedem regulären Ausdruck α entspricht eine Wortmenge $L(\alpha)$ aus Σ^*, die als *reguläre Menge* oder *reguläre Sprache* bezeichnet wird.

Beispiel 2.1. Zwei Beispiele von regulären Ausdrücken und deren Sprachen.

- Sei $\Sigma = \{a,b\}$, $\alpha = a^*ba$, dann ist $L(\alpha) = \{ba, aba, aaba, aaaba, \ldots\}$, die Menge aller Wörter, die mit beliebig vielen *a*s beginnen und mit *ba* endet.
- Sei $\Sigma = \{0,1\}, \alpha = (0|1)^*0(00|01|10|11)$, dann ist $L(\alpha) = \{x0y \mid x, y \in \Sigma^*, |y| = 2\}$, die Menge aller Bitfolgen mit einer 0 an drittletzter Stelle.

Ein regulärer Ausdruck kann also verstanden werden als *Formel*, die beschreibt, wie die Worte einer Sprache aus den Zeichen des Alphabets Σ und den genannten Operationen zu bilden sind. Ähnlich wie bei algebraischen Formeln gibt es vereinbarungsgemäß eine Operationen-Hierarchie: Es gilt, dass „*“ vor der Verkettung, und Verkettung vor „|“ bindet. Für die Verkettung gibt es kein spezielles Symbol, man notiert zwei Teilausdrücke einfach hintereinander. Man hat die Möglichkeit Klammern zu benutzen, wenn in einem Ausdruck eine andere Operation Vorrang haben soll.

Den Operationen Verkettung, Iteration und Auswahl zur Verknüpfung von regulären Ausdrücken entsprechen im Bereich der zugehörigen regulären Mengen aus Σ^* die Mengenoperationen *Mengen-Produkt, Iteration* und *Vereinigung*, die für Wortmengen U,V,W wie folgt definiert sind:

$$\begin{aligned}\text{Iteration:}\quad W^* &= \{\varepsilon\} \cup \{w_1 w_2 \dots w_n | w_i \in W \text{ für } i = 1,\dots,n \text{ und } n = 1,\dots,\infty\} \\ &= \{\varepsilon\} \cup W \cup WW \cup W^3 \cup W^4 \dots \\ &= \cup_{n=0}^{\infty} W^n\end{aligned}$$

$$\text{Produkt:}\quad UV = \{w = uv | u \in U \text{ und } v \in V\}$$

$$\text{Vereinigung:}\quad U \cup V = \{w | w \in U \text{ oder } w \in V\}$$

Definition 2.1. Formale Definition regulärer Ausdrücke und Sprachen. Es sei Σ ein endliches Alphabet. Wir definieren:

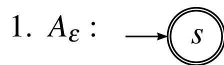
- ε ist ein regulärer Ausdruck mit der Sprache $L(\varepsilon) = \{\varepsilon\}$.
- Jedes $a \in \Sigma$ ist ein regulärer Ausdruck mit der Sprache $L(a) = \{a\}$.
- Mit den regulären Ausdrücken α und β sind auch α^*, $\alpha\beta$ und $\alpha|\beta$ reguläre Ausdrücke. Die zugehörigen Sprachen sind:
  - $L(\alpha^*) = L(\alpha)^*$
  - $L(\alpha\beta) = L(\alpha)L(\beta)$
  - $L(\alpha|\beta) = L(\alpha) \cup L(\beta)$

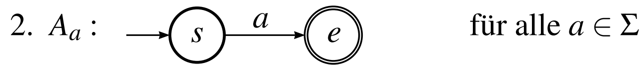
Häufig wird umgangssprachlich ein regulärer Ausdruck α mit seiner erzeugten Sprache $L(\alpha)$ gleichgesetzt. Wir wollen hier jedoch immer genau unterscheiden. Wir zeigen im Folgenden, dass reguläre Ausdrücke und endliche Automaten gleich mächtig sind.

2.2 Endliche Automaten und reguläre Sprachen

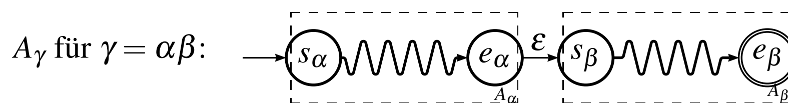
Satz 2.1. Zu jedem regulären Ausdruck α gibt es einen Automaten A mit genau einem Anfangs- und einem Endzustand, dessen Sprache $L(A)$ identisch ist mit der regulären Sprache $L(\alpha)$.

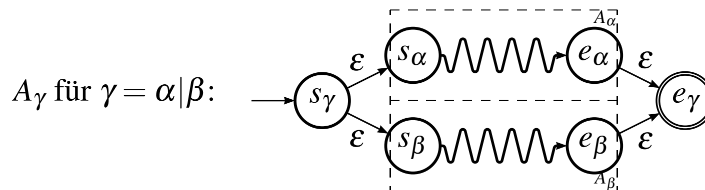
Beweis. Wir geben im Folgenden Elementarautomaten für die einfachen regulären Ausdrücke an und zeigen dann die Konstruktion von zusammengesetzten Automaten für beliebige reguläre Ausdrücke, gemäß Definitionen 2.1. Der konstruierte Automat ist ein nichtdeterministischer Automat mit ε-Übergängen.

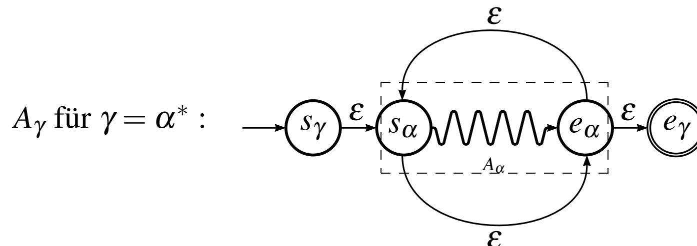




3. Seien A_α und A_β Automaten, die zu den regulären Ausdrücken α und β gehören, dann bilden wir für die Ausdrücke $\alpha\beta$, $\alpha|\beta$ und α^* zusammengesetzte Automaten nach folgendem Schema:







Dabei stehen s_α und s_β für die eindeutigen Anfangszustände und e_α und e_β für die eindeutigen Endzustände der Automaten A_α und A_β. Die Anfangs- und Endzustände sind eindeutig nach Voraussetzung. s_γ und e_γ stehen jeweils für neu einzuführende Zustände im zusammengesetzten Automat. Die ε-Übergänge wurden so gewählt, dass der zusammengesetzte Automat wieder eindeutige Anfangs- und Endzustände hat. □

Im Prinzip können die leicht verständlichen Schemata auch konstruktiv genutzt werden. In der Praxis kann es jedoch mühselig sein, die vielen ε-Übergänge erst aufzubauen und dann zu entfernen. Deshalb geben wir hier noch eine weitere Möglichkeit reguläre Operationen auf Automaten zu übertragen.

Definition 2.2. Konstruktion von zusammengesetzten Automaten.
Es seien $A_\alpha = (Z_\alpha, S_\alpha, F_\alpha, \Sigma, \delta_\alpha)$ und $A_\beta = (Z_\beta, S_\beta, F_\beta, \Sigma, \delta_\beta)$ die NEAs zu den regulären Ausdrücken α und β. Dabei steht Z für die Menge aller Zustände, S für die Menge der Anfangszustände und F für die Menge der Endzustände. Auf die Eigenschaft, dass es genau einen Anfangs- und einen Endzustand gibt, kann verzichtet werden.

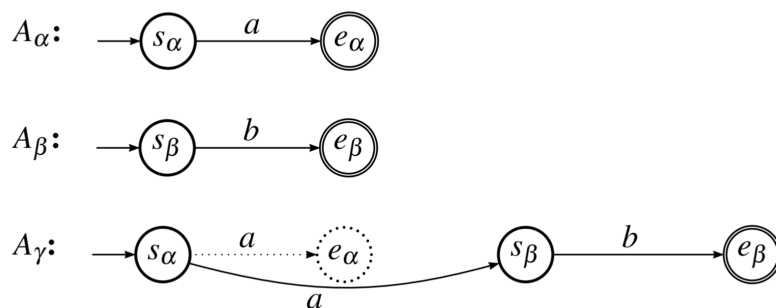
- Konstruktion des Automaten A_γ zu $\gamma = \alpha\beta$:

  Die Menge aller Zustände ist $Z_\gamma = Z_\alpha \cup Z_\beta$. Bei den Anfangszuständen beschränken wir uns auf $S_\gamma = S_\alpha$, falls A_α nicht das leere Wort akzeptiert. Anderenfalls benötigen wir $S_\gamma = S_\alpha \cup S_\beta$. Für die Endzustände gilt auf jeden Fall $F_\gamma = F_\beta$.

  Als Zustandsübergänge benötigen wir außer $\delta_\gamma = \delta_\alpha \cup \delta_\beta$ noch zusätzliche, die eine Verbindung von A_α nach A_β sicherstellen: Von jedem Zustand z_α, der einen Zustandsübergang $\delta_\alpha(z_\alpha, a) = e_\alpha$ in einen Endzustand e_α besitzt, muss auch ein Übergang $\delta_\gamma(z_\alpha, a) = s_\beta$ in jeden Anfangszustand s_β von A_β vorgesehen werden.

  Das Ergebnis ist ein NEA, in dem die Übergänge $\delta_\alpha(z_\alpha, a) = e_\alpha$ und damit auch die ehemaligen Endzustände e_α gegebenenfalls weggelassen werden können, wenn es sich um Senken des Zustandsgraphen handelt. Man kann auch aus dem erzeugten Automaten mit der Teilmengen-Konstruktion einen DEA erzeugen.

  Zum Beispiel sei $\Sigma = \{a, b\}$, $\alpha = a$, $\beta = b$ und $\gamma = \alpha\beta$.



  In diesem Beispiel können die gepunktet gezeichneten Elemente entfallen.

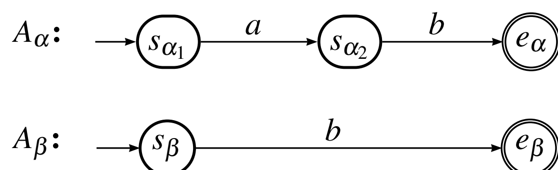
- Konstruktion des Automaten A_γ zu $\gamma = \alpha|\beta$:

  Für den „Oder"-Automaten A_γ gelten durchweg die Beziehungen:

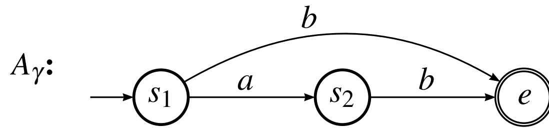
  $Z_\gamma = Z_\alpha \cup Z_\beta$, $S_\gamma = S_\alpha \cup S_\beta$, $F_\gamma = F_\alpha \cup F_\beta$, und $\delta_\gamma = \delta_\alpha \cup \delta_\beta$.

  Das Ergebnis ist ein NEA, der in der Regel auch reduzierbare Zustände besitzt.

  Zum Beispiel sei $\Sigma = \{a, b\}$, $\alpha = ab$, $\beta = b$, und $\gamma = \alpha|\beta$.



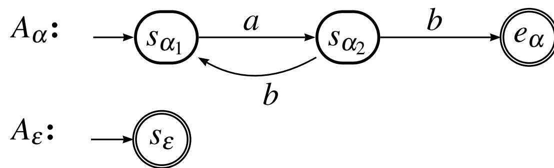
  Die beiden Automaten A_α und A_β bilden als Einheit den gesuchten Automat A_γ. Teilmengen-Konstruktion und Reduktion der Zustände ergeben den Minimalautomat.



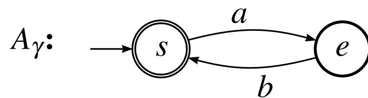
- Konstruktion des Automaten A_γ zu $\gamma = \alpha^*$:

  Alle Zustände werden in der Bedeutung wie im Automat zu α übernommen. Als Zustandsübergänge benötigen wir außer $\delta_\gamma = \delta_\alpha$ noch zusätzliche, die eine Verbindung vom Ende zum Anfang von A_α sicherstellen. Wir gehen dabei ähnlich wie bei $\gamma = \alpha\beta$ vor. Von jedem Zustand z_α, der einen Übergang $\delta_\alpha(z_\alpha, a) = e_\alpha$ in einen Endzustand e_α besitzt, ist ein Übergang $\delta_\gamma(z_\alpha, a) = s_\alpha$ in jeden Anfangszustand vorzusehen. Akzeptierte A_α bereits das leere Wort, dann sind wir fertig. Anderenfalls bilden wir noch den „Oder"-Automat mit dem Automat A_ε. Das Ergebnis ist wieder ein NEA.

  Zum Beispiel sei $S = \{a, b\}$, $\alpha = ab$, und $\gamma = \alpha^*$.



  Die beiden Automaten A_α und A_ε bilden als Einheit den gesuchten A_γ. Teilmengen-Konstruktion und Reduktion der Zustände ergeben den Minimalautomat für $(ab)^*$:



Es gilt auch die Umkehrung von Satz 2.1.

Satz 2.2. Zu jedem (determinierten) endlichen Automaten A über Σ kann man einen regulären Ausdruck α angeben mit $L(\alpha) = L(A)$.

Beweis. Sei $Z = \{z_0, z_1, \ldots, z_n\}$ die Menge der Zustände des Automaten A. Wir betrachten für je zwei Zustände z_i und z_l sowie leere Menge oder die Mengen $Z_k = \{z_0, z_1, \ldots, z_k\}$, $k = 0, 1, 2, \ldots, n$ folgende, wohldefinierte Ausdrücke:

$$\begin{aligned} w(z_i, \emptyset, z_l) &= \{w \in \Sigma^* \mid \delta(z_i, w) = z_l \text{ ohne Zwischenzustände } \} \\ w(z_i, Z_k, z_l) &= \{w \in \Sigma^* \mid \delta(z_i, w) = z_l \text{ und Zwischenzustände in } Z_k\} \end{aligned}$$

Für $w(z_i, \emptyset, z_l)$ lassen sich einfache reguläre Ausdrücke (Auswahl oder Iteration von Zeichen) angeben. Andererseits bestehen folgende, rekursiv anwendbare Zusammenhänge:

$$\begin{aligned} w(z_i, Z_0, z_l) &= w(z_i, \emptyset, z_l) | w(z_i, \emptyset, z_0)(w(z_0, \emptyset, z_0))^* w(z_0, \emptyset, z_l) \text{ für } k = 0 \\ w(z_i, Z_k, z_l) &= w(z_i, Z_{k-1}, z_l) | w(z_i, Z_{k-1}, z_k)(w(z_k, Z_{k-1}, z_k))^* w(z_k, Z_{k-1}, z_l) \\ &\quad \text{für } k > 0 \end{aligned}$$

Da jedes Wort, das zur Sprache $L(A)$ gehört, sich schreiben lässt als $w(z_0, Z_n, z_f)$ mit einem gewissen $z_f \in F = \{z_{f_1}, z_{f_2}, \ldots, z_{f_m}\}$, kann die Sprache $L(A)$ insgesamt beschrieben werden durch:

$$w(z_0, Z_n, z_{f_1})|w(z_0, Z_n, z_{f_2})|\ldots|w(z_0, Z_n, z_{f_m})$$

Daraus kann rekursiv ein regulärer Ausdruck α gebildet werden mit $L(\alpha) = L(A)$. □

Ähnlich zu dem Verfahren aus Definition 2.2 können wir ein Verfahren angeben, um einfacher und direkter reguläre Ausdrücke aus endlichen Automaten zu konstruieren.

Definition 2.3. Konstruktion eines regulären Ausdrucks aus einem endlichen Automaten.

- Eindeutiger Anfangszustand und eindeutiger Endzustand:

  Wir führen nach Lemma 1.2 durch neue ε-Übergänge einen eindeutigen Anfangszustand und einen eindeutigen Endzustand ein.

- Reguläre Ausdrücke an den Übergängen:

  Statt nur Zeichen oder ε an den Überführungen, lassen wir beliebige reguläre Ausdrücke an den Überführungen zwischen den Zuständen zu. Wir sehen ε und einzelne Zeichen $a \in \Sigma$ als reguläre Ausdrücke an. Mehrere Zeichen $a_1, a_2, \ldots, a_n$ an einem Übergang ersetzen wir durch den regulären Ausdruck $(a_1|a_2|\ldots|a_n)$.

- Elimination von Zuständen:

  Wir können einen Zustand s eliminieren, wenn alle möglichen Übergänge von beliebigen p_i über s nach q_j durch direkte Übergänge von p_i nach q_j mit entsprechenden regulären Ausdrücken ersetzt werden.

  Wir kommen zum Beispiel von p_1 nach q_1 indem wir

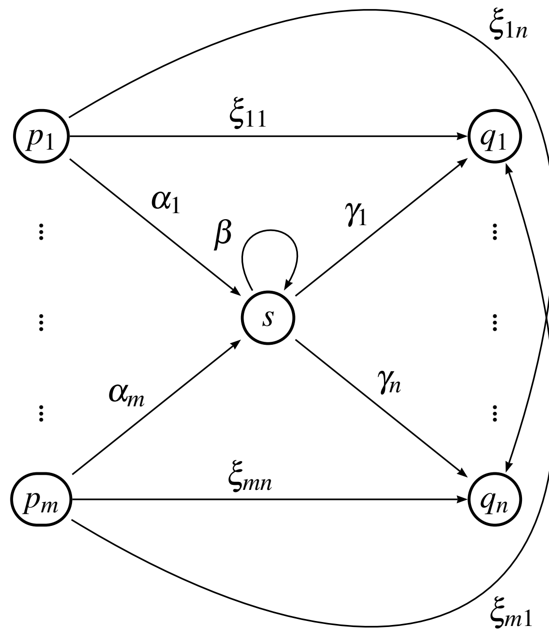
  - ξ_{11}: entweder direkt von p_1 nach q_1 gehen oder
  - $\alpha_1\beta^*\gamma_1$: von p_1 nach s gehen, beliebig häufig in s bleiben und dann von s nach q_1 gehen.

  Wir kommen also direkt von p_1 nach q_1 mit dem regulären Ausdruck

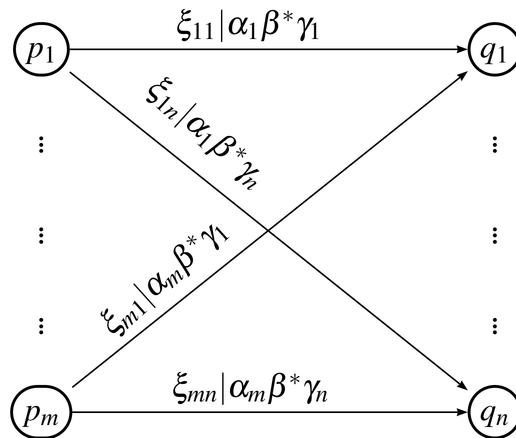
  $$\xi_{11}|(\alpha_1\beta^*\gamma_1)$$

  wobei alle möglichen Pfade durch s abgedeckt sind.

  Um s vollständig zu ersetzen, müssen wir jeden möglichen Pfad von allen p_i nach allen q_j für $1 \leq i \leq m, 1 \leq j \leq n$ durch den direkten Pfad mit dem regulären Ausdruck $\xi_{ij}|(\alpha_i\beta^*\gamma_j)$ ersetzen. Wir illustrieren dies in der folgenden Grafik und gehen aus von



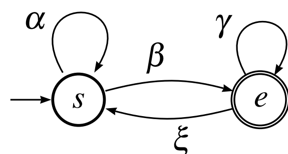
und erhalten Folgendes.



Der Zustand s ist eliminiert.

In obiger Abbildung steht ξ_{ij} für den regulären Ausdruck mit dem man direkt von p_i nach q_j gelangt, α_i für den mit dem man von p_i nach s gelangt, β für den mit dem man in s bleibt und γ_j für den mit dem man von s nach q_j gelangt.

- Eliminiere schrittweise alle Zustände außer dem Endzustand und dem Anfangszustand. Wir erhalten einen endlichen Automaten der Form



den wir direkt in den regulären Ausdruck

$$(\alpha^*|\beta\gamma^*\xi)^*\beta\gamma^*$$

übersetzen können.

Aus Satz 2.1 und Satz 2.2 können wir die Äquivalenz der Mächtigkeit von endlichen Automaten und regulären Ausdrücken folgern.

Satz 2.3. Die Familie der regulären Sprachen über einem endlichen Alphabet ist identisch mit der Menge der von endlichen Automaten akzeptierten Sprachen über diesem Alphabet.

2.3 Weitere Operationen und Abschlusseigenschaften regulärer Sprachen

Satz 2.4. Sind α und β reguläre Ausdrücke, das heißt also $L(\alpha)$ und $L(\beta)$ reguläre Sprachen, dann sind auch die folgenden Ausdrücke beziehungsweise Sprachen regulär:

- α^R: Spiegelung von α
- $\alpha^+ = \alpha\alpha^*$
- $\overline{L(\alpha)} = \Sigma^* \setminus L(\alpha)$: Komplement in Σ^*
- $L(\alpha) \cap L(\beta)$: Schnittmenge
- $L(\alpha) \setminus L(\beta)$: Differenzmenge

Zum *Beweis für die Spiegelung* (das Rückwärtsbearbeiten) eines Ausdrucks beachte man, dass alle elementaren regulären Ausdrücke gleich ihren gespiegelten sind und sich für einen zusammengesetzten Ausdruck ergibt:

$$\begin{aligned} (\alpha\beta)^R &= \beta^R\alpha^R, \\ (\alpha|\beta)^R &= \alpha^R|\beta^R, \\ (\alpha^*)^R &= (\alpha^R)^* \ . \end{aligned}$$

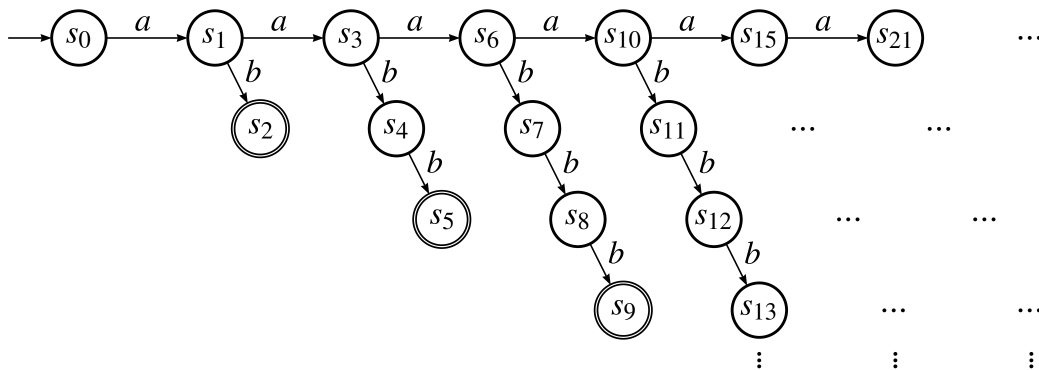
Zum *Beweis für die Komplementbildung* kann man auf den (vollständigen) determinierten Automaten zurückgreifen, der $L(\alpha)$ akzeptiert. Man konstruiert aus diesem durch Umdefinition der Endzustände in Nicht-Endzustände und umgekehrt einen Automaten, der nunmehr alle nicht zu $L(\alpha)$ gehörenden Worte akzeptiert.

2.4 Zwei Beispiele für nicht reguläre Sprachen

Die folgenden zwei Beispiele zeigen die *prinzipiellen Grenzen endlicher Automaten*.

Beispiel 2.2. Sei $\Sigma = \{a,b\}$ und $W = \{w \in \Sigma^* | w = a^m b^m, m = 1,2,\ldots\}$. Gibt es einen *endlichen* Automaten A mit $W = L(A)$?

Der Konstruktionsversuch



führt auf unendlich viele Zustände.

Beispiel 2.3. Sei $\Sigma = \{a,b\}$ und $W = \{w \in \Sigma^* | w = uu^R\}$, wobei u^R die Spiegelung von u ist. Beachten Sie, dass mit uu^R nicht der reguläre Ausdruck im Sinne von $L(u)L(u^R)$ gemeint ist. Gibt es einen *endlichen* Automaten A mit $W = L(A)$?

Die Antwort ist „Nein“. Die Konstruktion eines entsprechenden Automaten scheitert daran, dass beim endlichen Automaten nicht festgestellt werden kann, wann die Mitte eines beliebig langen Wortes uu^R erreicht ist.

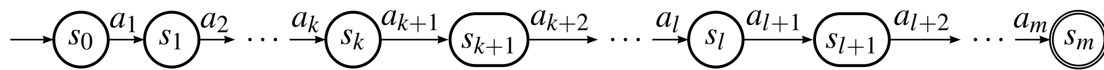
Im Allgemeinen können wir feststellen: Endliche Automaten sind nicht in der Lage, eingelesene Zeichen dynamisch zu speichern. Es fehlt ihnen eine Speicher-Einrichtung. Sie können deshalb nicht vergleichen, messen und so weiter.

2.5 Das Pumping-Lemma für reguläre Sprachen

Satz 2.5. Ist n die Anzahl der Zustände eines endlichen Automaten und besteht ein Wort $w \in L(A)$ aus n Zeichen oder mehr (also $|w| \geq n$), dann kann w zerlegt werden in $w = xyz$, wobei für die Komponenten Folgendes gilt:

$$\begin{aligned} |xy| &\leq n, \\ |y| &\geq 1 \text{ und} \\ xy^k z &\in L(A) \text{ für } k = 0,1,2,\ldots \end{aligned}$$

Beweis. Es sei



die Folge der Zustände, die beim Abarbeiten des Wortes $w = a_1a_2 \dots a_m$ mit der Länge $m \geq n$ auftreten. Nach dem Abarbeiten der ersten n Zeichen muss mindestens ein Zustand zweimal aufgetreten sein. Wir nehmen an, dass dies beim k-ten und l-ten Arbeitsschritt mit $k < l \leq n$ der Fall war, so dass also $s_k = s_l$ gilt. Wir wählen:

$$\begin{aligned} x &= a_1 \dots a_k, \\ y &= a_{k+1} \dots a_l \text{ und} \\ z &= a_{l+1} \dots a_m \ . \end{aligned}$$

Dann gilt offensichtlich $w = xyz$, $1 \leq |y|$ und $|xy| \leq n$. Da außerdem $s_k = s_l$ ist, kann nach dem Teilwort x unmittelbar das Teilwort z folgen, als auch das Teilwort y beliebig oft wiederholt werden. □

Das Pumping-Lemma, auch *Schleifensatz* genannt, wird häufig zum Nachweis, dass eine formale Sprache nicht als Sprache eines endlichen Automaten aufgefasst werden kann, herangezogen.

Beispiel 2.4. Gegeben sei die Sprache $L = \{a^m b^m \mid m = 1, 2, \dots\}$ (oder kurz $a^m b^m$), also alle Wörter mit der gleichen Anzahl an Zeichen a am Anfang wie Zeichen b am Ende. Wir zeigen, dass die Sprache L nicht regulär ist.

Dazu nehmen wir erst einmal an, dass $a^m b^m$ regulär sei. Sei nun n die Anzahl der Zustände des endlichen Automaten A, der Worte der Form $a^m b^m$ akzeptiert, also das n aus dem Pumping-Lemma, Satz 2.5. Wir wählen das Wort $w = a^n b^n$. Offensichtlich gilt, dass w die Form $a^m b^m$ hat. Laut Pumping-Lemma gibt es eine Zerlegung von w in xyz für die gilt:

$$\begin{aligned} |xy| &\leq n, \\ |y| &\geq 1 \text{ und} \\ xy^k z &\in L(A) \text{ für } k = 0, 1, 2, \dots \end{aligned}$$

Da $|xy| \leq n$ wissen wir, dass xy nur aus a's bestehen kann. Also besteht auch y nur aus a's. Laut Pumping-Lemma ist für alle $k = 0, 1, 2, \dots$ das Wort $xy^k z$ auch in der Sprache. Sei $k = 2$. Da $|y| \geq 1$ fügen wir dem Wort $a^n b^n$ neue a's hinzu und erhalten das Wort $xy^2 z = a^{n+|y|} b^n$. Offensichtlich ist $xy^2 z$ aber *nicht* von der Form $a^m b^m$. Wir erhalten einen Widerspruch.

Aus dem Widerspruch folgern wir, dass unsere einzige Annahme falsch sein muss. Die Sprache $a^m b^m$ ist also *nicht* regulär.

Kapitel 3

Grammatiken formaler Sprachen

Mit den bisher behandelten Automaten wurden Sprachen nur indirekt beschrieben. Sie sind mit den Automaten als Akzeptoren identifizierbar aber nicht direkt erzeugbar. Schon eher kann man die regulären Ausdrücke als eine Methode zur Erzeugung von Sprachen ansehen. Im Folgenden werden wir noch andere Erzeugungssysteme (Grammatiken) für eine größere Klasse von Sprachen kennen lernen. Die Worterzeugung beruht dabei auf dem Prinzip, in einem vorhandenen Wort ein Teilwort durch ein anderes zu ersetzen.

3.1 Semi-Thue-Systeme

Semi-Thue-Systeme stellen die *allgemeinste* Form der Regelsysteme für formale Sprachen dar und werden hier nur kurz behandelt. Sie sind nach dem norwegischen Mathematiker A. Thue benannt, der 1914 eine Arbeit darüber veröffentlichte.

Definition 3.1. Ein *Semi-Thue-System* besteht aus einem endlichen Alphabet Σ und einer endlichen Menge von Wortpaaren $\{(\alpha,\beta) \mid \alpha,\beta \in \Sigma^*\}$. Jedes Wortpaar ist eine Regel in dem Sinne, dass in einem vorhandenen Ausgangswort w ein Teilwort α durch β ersetzt werden kann[1].

Definition 3.2. Ein Wort w heißt aus einem Wort v *ableitbar*, wenn es durch endlich viele Ersetzungsschritte aus v entsteht, in Zeichen:

$$v \Rightarrow^* w : v = v_0 \stackrel{\alpha_0 \to \beta_1}{\Longrightarrow} v_1 \stackrel{\alpha_1 \to \beta_2}{\Longrightarrow} \dots v_{n-1} \stackrel{\alpha_{n-1} \to \beta_n}{\Longrightarrow} v_n = w$$

Beispiel 3.1. Sei $\Sigma = \{a,b,c,d,e\}$ und die Ersetzungsregeln wie folgt definiert

$$\begin{array}{ll} (1) & ab \to ad \\ (2) & dc \to ee \end{array}$$

[1] Die einseitige Ersetzungsrichtung $\alpha \to \beta$ erklärt die Bezeichnung *Semi-*....

$$\begin{array}{ll} (3) & e \rightarrow b \\ (4) & ad \rightarrow ae \\ (5) & eb \rightarrow b \\ (6) & abc \rightarrow e \end{array}$$

Wie zeigen die Ableitung von $v = abc$ nach $w = aeb$

$$v \Rightarrow^* w : a\underline{bc} \stackrel{(1)}{\Longrightarrow} a\underline{dc} \stackrel{(2)}{\Longrightarrow} ae\underline{e} \stackrel{(3)}{\Longrightarrow} aeb = w$$

Satz 3.1 (Allgemeines Wortproblem)**.** Die Frage, ob ein Wort w aus einem Wort v abgeleitet werden kann, ist für beliebige Semi-Thue-Systeme nicht entscheidbar.

Wenn das Wortproblem nicht entscheidbar ist, dann bedeutet das, dass es keinen Algorithmus gibt, der bei gegebenem v, w und gegebenen Ersetzungsregeln feststellen kann, ob es eine Ableitung von w aus v gibt. Wir werden uns in Kapitel 9 noch im Detail mit Entscheidbarkeit auseinandersetzen.

3.2 Chomsky-Grammatiken

Die Chomsky-Grammatiken sind nach dem amerikanischen Sprachwissenschaftler Noam Chomsky benannt, der sich in den fünfziger Jahren mit diesem Thema beschäftigte. Ein wesentlicher Unterschied zur vorhergehenden Betrachtungsweise ist, dass innerhalb Σ nach *Terminal-* und *Nicht-Terminalzeichen* unterschieden wird und die zu erzeugenden Worte letztendlich aus Terminalzeichen bestehen.

Definition 3.3. $G = (N,T,P,S)$ ist eine *Chomsky-Grammatik*, wenn die einzelnen Komponenten folgende Bedeutung haben:

N Endliche Menge der Nicht-Terminalsymbole $A,B,C,\ldots$

T Endliche Menge der Terminalsymbole $a,b,c,\ldots$

P Endliche Menge der Produktionen (Regeln): $P \subset \{\alpha \rightarrow \beta | \alpha, \beta \in (N \cup T)^*, \alpha \neq \varepsilon\}$

S Startsymbol in N, das in mindestens einer Regel links vorkommt

Die Produktionen P stellen also ein Semi-Thue-System über $\Sigma^* = (N \cup T)^*$ mit besonderen Eigenschaften dar.

Definition 3.4. $L(G)$ ist die von einer Chomsky-Grammatik G *erzeugte Sprache*. $L(G)$ besteht aus allen Worten über T^*, die aus S ableitbar sind, das heißt

$$L(G) = \{w \in T^* | S \Rightarrow^* w\} \quad .$$

Beispiel 3.2. Erzeugen von Fließkommazahlen.

$N = \{S, V, I, D\}$, $V =<$ Vorzeichen $>$, $I =<$ Integerzahl $>$, $D =<$ Digit $>$

$T = \{+, -, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, e\}$

$P =$	$(1) S \to VI$	$(5) I \to D$	$(7) V \to +$
	$(2) S \to VI.I$	$(6) I \to DI$	$(8) V \to -$
	$(3) S \to VIeVI$		$(9) V \to \varepsilon$
	$(4) S \to VI.IeVI$	$(10) D \to 0\|1\|2\|3\|4\|5\|6\|7\|8\|9$	

3.3 Die Chomsky-Hierarchie

Eine Chomsky-Grammatik heißt vom

Typ 3 (oder rechtslinear): wenn alle Regeln die Form $A \to aB$, $A \to a$, $A \to B$ oder $A \to \varepsilon$ haben.

Typ 2 (oder kontextfrei): wenn alle Regeln die Form $A \to \gamma$ haben.

Typ 1 (monoton oder kontextsensitiv):

monoton: wenn alle Regeln die Form $\alpha \to \beta$ mit $|\alpha| \leq |\beta|$ haben.

kontextsensitiv: wenn alle Regeln die Form $\alpha A \beta \to \alpha \gamma \beta$ mit $\gamma \neq \varepsilon$ haben.

Typ 0: wenn alle Regeln die *nicht eingeschränkte* Form $\alpha \to \beta$ haben.

Dabei gilt $\alpha, \gamma, \beta \in (N \cup T)^*$. Bei einer Grammatik vom Typ 1 kann man die Regel $S \to \varepsilon$ zulassen, wenn S in keiner anderen Regel auf der rechten Seite vorkommt.

Definition 3.5. Zwei Grammatiken G_1 und G_2 heißen *äquivalent*, wenn sie die gleiche Sprache erzeugen ($L(G_1) = L(G_2)$).

Definition 3.6. Eine Sprache $L(G)$ heißt vom *Chomsky-Typ i* mit $(i = 0, 1, 2, 3)$ wenn G vom Typ i ist. Die Familie der Sprachen vom Typ i wird mit L_i bezeichnet.

Es gelten die folgenden Beziehungen:

- $L_3, L_2, L_1 \subset L_0$
- $L_3 \subset L_2$
- $L_2 \subset L_1$
- Die Familie der kontextsensitiven Sprachen ist gleich der Familie der monotonen Sprachen.

Beweise dazu, beziehungsweise Erläuterungen, finden Sie in den folgenden Abschnitten.

3.4 Endliche Automaten und rechtslineare Grammatiken

Die Regeln der rechts- oder linkslinearen Grammatiken sind innerhalb der Chomsky-Hierarchie am stärksten eingeschränkt: Es kann ein Nicht-Terminalzeichen nur durch ein Terminalzeichen und gegebenenfalls wieder ein Nicht-Terminalzeichen ersetzt werden. Bei Rechtslinearität erfolgt diese Ersetzung immer rechts, am Ende der bereits vorhandenen Terminalzeichenkette, so dass das entstehende Wort in einer Richtung *(linear)* mit der Anzahl der Ersetzungsschritte wächst.

Satz 3.2. Zu jeder rechtslinearen Grammatik gibt es eine linkslineare, die die gleiche Sprache erzeugt.

Satz 3.3. Zu jeder rechtslinearen Grammatik mit der Sprache $L(G)$ gibt es einen endlichen Automaten A mit $L(A) = L(G)$ und umgekehrt.

Beweis. Sei $A = (S, s_0, F, \Sigma, \delta)$ ein endlicher Automat und $G = (N, T, P, S)$ eine rechtslineare Grammatik.
$A \rightsquigarrow G$: Konstruiere aus einem endlichen Automaten A eine rechtslineare Grammatik G.

- Verwende die Eingabezeichen Σ als Terminalzeichen T
- Verwende die Menge S der Zustände als Nicht-Terminalzeichen N
- Verwende s_0 als Startsymbol S
- Erzeuge für jeden Funktionswert $\delta(s_1, a) = s_2$ die Regel $s_1 \to as_2$
- Erzeuge für jeden Endzustand s_e eine zusätzliche Regel $s_e \to \varepsilon$

$G \rightsquigarrow A$: Konstruiere aus einer rechtslinearen Grammatik G einen endlichen Automaten A.

- Verwende die Terminalzeichen T als Eingabezeichen Σ
- Verwende die Nicht-Terminalzeichen N als Zustände S
- Verwende das Startsymbol S als Ausgangszustand s_0
- Verwende jedes A, für das $A \to \varepsilon$ gilt, als Endzustand
- Definiere für jede Produktion $A \to aB$ den Funktionswert $\delta(A, a) = B$
- Führe für *alle* Produktionen $A \to a$ *einen* Endzustand s_{e_a} ein und bilde für *jede* solche Produktion den Funktionswert $\delta(A, a) = s_{e_a}$

□

Aus Satz 3.3 folgt, dass wir mit rechtslinearen Grammatiken genau die regulären Sprachen erzeugen können.

Satz 3.4. Die Menge der Sprachen, die durch rechtslineare Grammatiken erzeugt werden, ist identisch mit der Menge der regulären Sprachen.

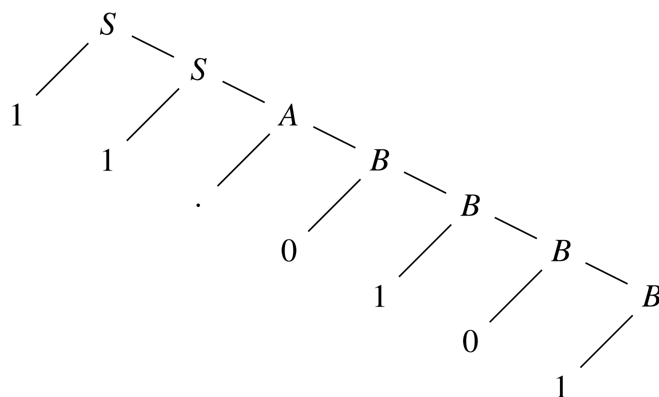
Wir können also die Sprache $a^n b^n$ *nicht* durch eine rechtslineare Grammatik erzeugen.

Definition 3.7. Man versteht unter einem *Ableitungsbaum* oder *Parsebaum* einen Graph, der die Ableitungsfolge bei der Erzeugung eines Worts bildlich darstellt.

Beispiel 3.3. Eine rechtslineare Grammatik für Dualzahlen mit oder ohne Nachkommastellen ist gegeben durch folgende Regeln:

$$\begin{aligned} S &\rightarrow 0S|1S|0A|1A \\ A &\rightarrow \varepsilon|.B \\ B &\rightarrow 0B|1B|0|1 \end{aligned}$$

Der Ableitungsbaum des Wortes 11.0101 ist also wie folgt:



Das bereits bekannte Pumping-Lemma für reguläre Sprachen, Satz 2.5, lässt sich hier noch einmal anders formulieren und gut nachvollziehen.

Satz 3.5. Es sei $L(G)$ die Sprache einer rechtslinearen Grammatik G. Dann kann jedes Wort $w \in L(G)$ mit einer Länge $|w| > |N|$, wobei $|N| = n =$ Anzahl der verschiedenen Nicht-Terminalzeichen in G, zerlegt werden in $w = xyz$, wobei gilt

- $|xy| \leq n$,
- $|y| \geq 1$ und
- $w = xy^i z \in L(G)$ für $i = 0, 1, 2 \ldots$.

Beweis. Man betrachte den Ableitungsbaum von w für die ersten $n+1$ Ableitungsschritte. Dabei wird $n+1$ mal ein Nicht-Terminalsymbol benötigt. Wenigstens ein Nicht-Terminalsymbol A muss an mindestens zwei verschiedenen Stellen innerhalb dieser $n+1$ Schritte

auftauchen. Wir bezeichnen mit y das Teilwort, das vom erstmaligen Auftauchen von A bis zum nächsten Auftauchen von A abgeleitet werden kann, mit x das Teilstück, das y vorangestellt und mit z das Teilstück, das y nachfolgt. Dann gilt offenbar $|xy| \leq n$ und $|y| \geq 1$. Außerdem kann das Teilstück y im Ableitungsbaum beliebig oft wiederholt werden. Also gilt auch der dritte Teil der Behauptung. □

Kapitel 4

Kontextfreie Sprachen

Die Regelform $A \to w$ kontextfreier Grammatiken, wobei A ein Nicht-Terminalzeichen und w eine Zeichenfolge aus Nicht-Terminalzeichen und Terminalzeichen sein kann, ist charakteristisch für die meisten höheren Programmier- und Auszeichnungssprachen. Die Regeln für Programmiersprachen liegen meist in (erweiterter) Backus-Naur-Form (E)BNF oder Syntaxdiagrammen vor und können leicht in kontextfreie Grammatiken überführt werden.

Beispiel 4.1. Die Darstellung einer Gleitkommazahl ohne führende Nullen in EBNF ist

```
Gleitkommazahl ::= [ Vorzeichen ] SigZahl [ '.' Zahl ]
Vorzeichen ::= '+' | '−'
SigZahl ::= ZifferAusserNull { Ziffer } | '0' ;
ZifferAusserNull ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Ziffer ::= '0' | ZifferAusserNull
Zahl ::= Ziffer { Ziffer };
```

Ein umgesetzte Grammatik ergibt sich wie folgt:

$$
\begin{array}{lllllllll}
S & \to & VAB & & & & & & \\
V & \to & + & V & \to & - & V & \to & \varepsilon \\
A & \to & YD & A & \to & 0 & & & \\
D & \to & ZD & D & \to & \varepsilon & & & \\
B & \to & .E & B & \to & \varepsilon & & & \\
E & \to & ZD & & & & & & \\
Y & \to & 1|2|3|4|5|6|7|8|9 & & & & & & \\
Z & \to & 0|1|2|3|4|5|6|7|8|9 & & & & & &
\end{array}
$$

Dabei steht S für die Gleitkommazahl, V für das optionale Vorzeichen, A für SigZahl und B für den optionalen Dezimalpunkt und nachfolgende Zahl. Mit D werden beliebig lange Ziffernfolgen inklusive des leeren Wortes und mit E beliebig lange Ziffernfolgen mit mindestens einer Ziffer erzeugt. Y erzeugt schließlich eine Ziffer außer der 0, während Z eine beliebige Ziffer inklusive der 0 erzeugt. Wir verwenden hier $A \to B|C$ als Abkürzung für die beiden Produktionen $A \to B$ und $A \to C$.

Wir können zum Beispiel die Gleitkommazahl 1.23 herleiten mit

$$S \Rightarrow VAB \Rightarrow AB \Rightarrow YDB \Rightarrow 1DB \Rightarrow 1B \Rightarrow 1.E \Rightarrow 1.ZD \Rightarrow 1.2D \Rightarrow 1.2ZD \Rightarrow 1.23D \Rightarrow 1.23$$

Beispiel 4.2. Ein typisches Beispiel für kontextfreie Grammatiken ist die Definition von Grammatiken für beliebig geklammerte Ausdrücke.

$$\begin{array}{rcl} E & \rightarrow & T|E+T \\ T & \rightarrow & F|T*F \\ F & \rightarrow & a|(E) \end{array}$$

Dabei ist a ein Platzhalter für Variablen oder Werte in einer (Programmier-)Sprache. Es steht E für Ausdruck (Expression), T für Term und F für Funktor.

Während die Gleitkommazahlen aus dem Beispiel 4.1 noch durch reguläre Ausdrücke beschreibbar sind, ist dies für die geklammerten Ausdrücke aus Beispiel 4.2 nicht mehr möglich. Dass wir uns außerhalb der Sprachklasse der regulären Sprachen befinden, sehen wir am nächsten Beispiel.

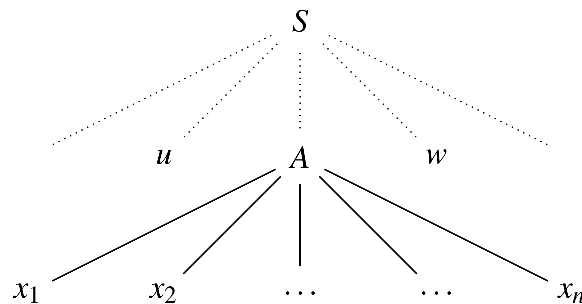
Beispiel 4.3. Eine kontextfreie Grammatik für die Sprache $a^n b^n$.

$$\begin{array}{rcl} N & = & \{S\} \\ T & = & \{a,b\} \\ P & = & \{S \rightarrow aSb, S \rightarrow \varepsilon\} \end{array}$$

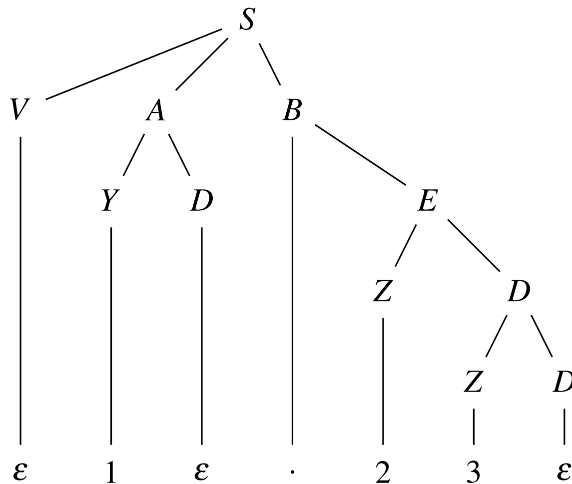
Ableitung von a^3b^3: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$

Wesentlich für die Mächtigkeit sind wie bei allen Grammatiken die rekursiven Regeln, wobei sich eine Rekursion auch über mehrere Regeln erstrecken kann. Bei den kontextfreien Grammatiken können außer linksrekursiven Regeln (Typ: $A \rightarrow Aw$) und rechtsrekursiven Regeln (Typ: $A \rightarrow wA$) nunmehr auch zentralrekursive Regeln (Typ: $A \rightarrow vAw$) beziehungsweise zentrale Rekursionen vorkommen. Dies ist eine spezifische Eigenschaft kontextfreier Grammatiken.

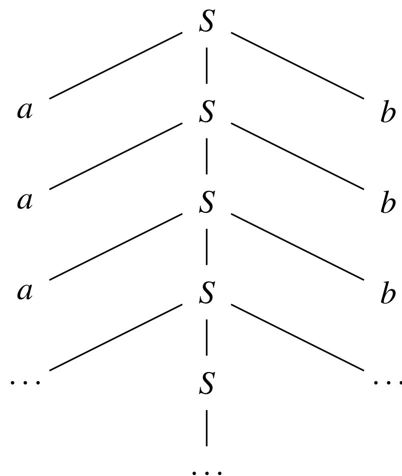
Definition 4.1. Ein *Ableitungsbaum* bei kontextfreien Grammatiken ist eine graphische Darstellung der Ableitungsschritte. Es sei ein Wort uAw schon teilweise abgeleitet. Wir wollen die Regel $A \rightarrow v$ anwenden. Dabei sei im Allgemeinen $v \in (N \cup T)^*$ von der Form $x_1 x_2 \ldots x_n$. Dann erhalten wir den folgenden Ableitungsbaum.



Beispiel 4.4. Ableitungsbaum von Beispiel 4.1, den Gleitkommazahlen.



Beispiel 4.5. Ableitungsbaum von Beispiel 4.3, $a^n b^n$.

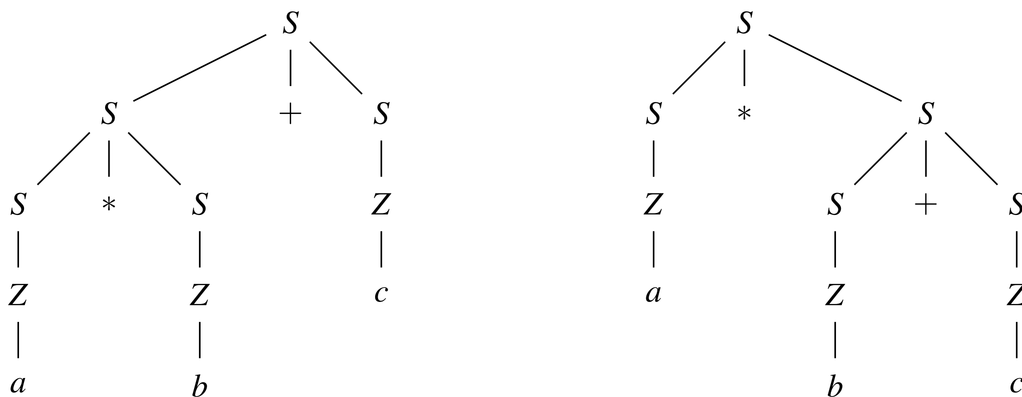


4.1 Mehrdeutigkeit bei kontextfreien Grammatiken

Beispiel 4.6. Grammatik für einfache arithmetische Ausdrücke.

$$\begin{array}{lcl} T & = & \{a,b,c,+,*,(,)\} \\ N & = & \{S,Z\} \\ P & = & \{S \to S+S|S*S|(S)|Z, \quad Z \to a|b|c\} \end{array}$$

Für die Ableitung von $a*b+c$ gibt es zwei mögliche Ableitungsbäume. Beachten Sie, dass die Rechenregel „Multiplikation vor Addition“ in den Ableitungsregeln rechts ignoriert wird.



Definition 4.2. Eine Grammatik heißt *mehrdeutig*, wenn es ein Wort in $L(G)$ gibt, zu dem unterschiedliche Ableitungsbäume existieren.

Der Ableitungsbaum ist ein statisches Gebilde, zu dem man auf verschiedenen Wegen, das heißt durch unterschiedliche Reihenfolgen der Ableitungsschritte kommen kann.

Definition 4.3. Eine Ableitung heißt *linksseitige* Ableitung, wenn in einer Ableitungsfolge immer das äußerste linke Nicht-Terminalzeichen ersetzt wird. Wir schreiben $v \Rightarrow_{lm}^{*} w$ für eine linksseitige Ableitung von v nach w. Analog dazu kann man auch eine *rechtsseitige* Ableitung ($\Rightarrow_{rm}^{*}$) definieren[1].

Satz 4.1. Für jede Ableitung gibt es eine äquivalente linksseitige und eine äquivalente rechtsseitige Ableitung. Es gilt, dass $v \Rightarrow^{*} w$ genau dann, wenn $v \Rightarrow_{lm}^{*} w$ genau dann, wenn $v \Rightarrow_{rm}^{*} w$.

Dies ist wichtig für konkrete Anwendungen wie Parser, die Ersetzungsschritte in einer fest vorgegebenen Reihenfolge ausführen wollen.

Beispiel 4.7. Für die Herleitung des linken Ableitungsbaums von $a*b+c$ aus Beispiel 4.6 gibt es eine äquivalente linksseitige

$$\begin{aligned} S &\Rightarrow_{lm} S+S \Rightarrow_{lm} S*S+S \Rightarrow_{lm} Z*S+S \Rightarrow_{lm} a*S+S \Rightarrow_{lm} a*Z+S \\ &\Rightarrow_{lm} a*b+S \Rightarrow_{lm} a*b+Z \Rightarrow_{lm} a*b+c \end{aligned}$$

und rechtsseitige

$$\begin{aligned} S &\Rightarrow_{rm} S+S \Rightarrow_{rm} S+Z \Rightarrow_{rm} S+c \Rightarrow_{rm} S*S+c \Rightarrow_{rm} S*Z+c \\ &\Rightarrow_{rm} S*b+c \Rightarrow_{rm} Z*b+c \Rightarrow_{rm} a*b+c \end{aligned}$$

Ableitung. Es wird der gleiche Ableitungsbaum erzeugt.

[1] *lm* steht für Englisch „leftmost“ und *rm* für Englisch „rightmost“.

Wenn der Ableitungsbaum immer gleich ist, bedeutet dies also noch keine Mehrdeutigkeit der Grammatik.

Beispiel 4.8. Eindeutige Grammatik, die äquivalent zu Beispiel 4.6 ist.

$$\begin{aligned} T &= \{a, b, c, +, *, (,)\} \\ N &= \{S, T, F, Z\} \\ P &= \{S \to T|S+T, \quad T \to F|F*T, \quad F \to Z|(S), \quad Z \to a|b|c\} \end{aligned}$$

Mehrdeutige Grammatiken sind unerwünscht, weil sie Interpretationsschwierigkeiten verursachen können. Mehrdeutigkeit kann aber unter Umständen nicht vermieden werden. Man spricht von einer *inhärent mehrdeutigen Sprache*, wenn jede Grammatik, die die Sprache erzeugt, mehrdeutig ist.

Beispiel 4.9. Eine inhärent mehrdeutigen Sprache.

$$L = \{a^l b^m c^n | l = m \text{ oder } m = n \text{ und } l, m, n = 0, 1, 2, \ldots\} = L_1 \cup L_2$$

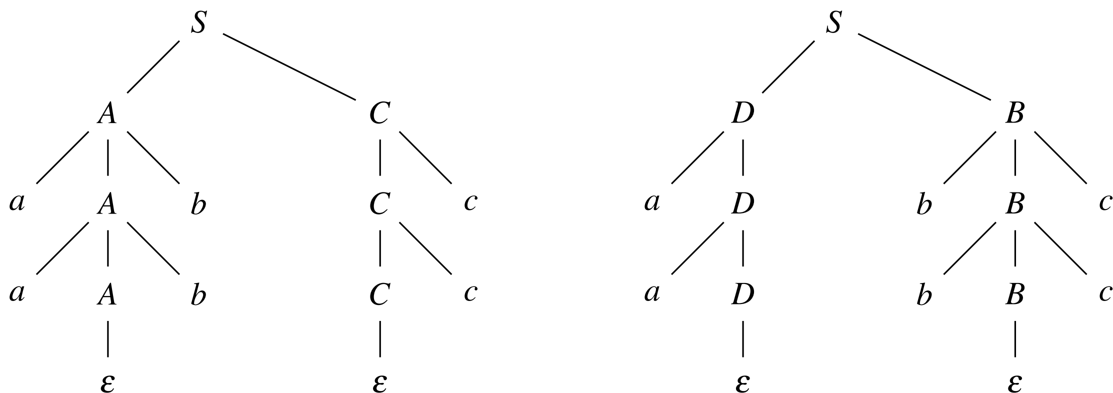
mit

$$L_1 = \{a^m b^m c^n | m, n = 0, 1, 2, \ldots\} \text{ und } L_2 = \{a^m b^n c^n | m, n = 0, 1, 2, \ldots\}$$

Jede Grammatik G für L setzt sich im Wesentlichen zusammen aus den beiden zu L_1 und L_2 gehörigen Grammatiken G_1 und G_2.

$$\begin{aligned} G_1: \quad T &= \{a, b, c\} & N &= \{A, S, C\} \\ P &= \{S \to AC, \quad A \to aAb|\varepsilon, \quad C \to cC|\varepsilon\} \\ G_2: \quad T &= \{a, b, c\} & N &= \{D, S, B\} \\ P &= \{S \to DB, \quad D \to aD|\varepsilon, \quad B \to bBc|\varepsilon\} \end{aligned}$$

Für ein Wort $a^m b^m c^m$ ergeben sich deshalb immer zwei Möglichkeiten, zum Beispiel für $m = 2$:



Man kann zeigen, dass es keine Grammatik für L gibt, die einen eindeutigen Syntaxbaum liefert.

4.2 Normalformen kontextfreier Grammatiken

Kontextfreie Grammatiken mit Produktionsregeln beliebiger Form machen es schwer, Aussagen über erzeugte Sprachen zu machen und diese Grammatiken zum Erkennen von Wörtern zu verwenden. Wir schränken deshalb die Arten der Produktionsregeln weiter ein und definieren Normalformen von kontextfreien Grammatiken, mit denen man besser arbeiten kann, die aber immer noch alle kontextfreien Sprachen erzeugen können.

Definition 4.4. Eine kontextfreie Grammatik heißt *ε-frei*, falls keine Produktion $A \to \varepsilon$ vorhanden ist.

Satz 4.2. Zu jeder kontextfreien Grammatik G mit der Sprache $L(G)$ gibt es eine ε-freie Grammatik G' mit $L(G') = L(G) - \{\varepsilon\}$.

Um aus einer Grammatik eine ε-freie Grammatik zu machen, müssen wir zuerst alle Nicht-Terminalzeichen A finden, die ε erzeugen, für die also gilt $A \Rightarrow^* \varepsilon$. Wir konstruieren die Menge

$$V = \{A \mid A \Rightarrow^* \varepsilon\}$$

wie folgt. Initial besteht V aus allen Nicht-Terminalzeichen A, für die $A \to \varepsilon$ in P ist. Wir fügen ein Nicht-Terminalzeichen B zu V hinzu falls $B \to A_1 \dots A_n$ in P ist und alle A_i für $1 \leq i \leq n$ der rechten Seite der Regel schon in V sind. Wir wiederholen den Vorgang so lange, bis keine weiteren Nicht-Terminalzeichen zu V hinzugefügt werden können. Die Menge der Nicht-Terminalzeichen in V heißt *eliminierbar*.

Wir können jetzt die Produktionen $A \to \varepsilon$ aus P entfernen. Damit aber immer noch alle Ableitungen möglich sind, müssen wir die Satzformen, die ein $A \in V$ auf ε ableiten vorweg nehmen. Wir fügen also sukzessive für alle Produktionen $B \to uAv \in P$ mit $B \in N$, $A \in V$ und $u, v \in (N \cup T)^*$ sowie $|uv| \geq 1$ die Produktion $B \to uv$ zu P hinzu. Beachten Sie, dass alle Möglichkeiten B, u, A, v zu wählen berücksichtigt werden müssen. So können aus neu hinzugefügten Produktionen eventuell wieder weitere Produktionen entstehen.

Beispiel 4.10. Wir haben folgende fünf Produktionen.

$$\begin{array}{lcl} S & \to & AB \\ A & \to & aAA \mid \varepsilon \\ B & \to & bBB \mid \varepsilon \end{array}$$

Zunächst bestimmen wir V. Die Nicht-Terminalzeichen A und B sind in V, da $A \to \varepsilon$ und $B \to \varepsilon$ in P sind. Da sowohl A und B in V sind und damit alle Zeichen der rechten Seite der Regel $S \to AB$, ist auch S in V.

Wir müssen nun sukzessive alle Vorkommen von S,A oder B auf rechten Seiten eliminieren. Wir erhalten als Menge von Produktionen:

$$\begin{array}{lcl} S & \to & AB \mid A \mid B \\ A & \to & aAA \mid aA \mid a \\ B & \to & bBB \mid bB \mid b \end{array}$$

Beachten Sie, dass $S \Rightarrow^* \varepsilon$ nicht mehr herleitbar ist.

Falls ε ein Wort einer Sprache $L(G)$ ist, dann können wir zu der ε-freien Grammatik zwei neue Produktionen hinzunehmen

$$S' \to \varepsilon \qquad S' \to S$$

wobei S' ein neues Nicht-Terminalzeichen und gleichzeitig neues Startsymbol ist. S' darf in keiner weiteren Regel mehr auftauchen. Die einzige Aufgabe der beiden neuen Regeln ist es, das leere Wort zu erzeugen. Die neue Grammatik erzeugt dann dieselbe Sprache, inklusive leerem Wort, wie die ursprüngliche Grammatik vor der ε-Eliminierung.

Definition 4.5. Ein Nicht-Terminalzeichen A heißt *nützlich*, wenn es eine Ableitung

$$S \Rightarrow^* uAv \Rightarrow^* w$$

gibt mit $w \in T^*$ und $u, v \in (N \cup T)^*$. Das heißt also, wenn A einerseits vom Startsymbol erreicht wird (*erreichbar*) und schließlich ein Wort der Sprache herleitbar ist (*erzeugend*) .

Satz 4.3. Zu jeder kontextfreien Grammatik G gibt es eine kontextfreie Grammatik G', deren Produktionsregeln nur nützliche Nicht-Terminalzeichen beinhaltet, mit $L(G') = L(G)$.

Aus einer Grammatik können alle Regeln, die entweder auf der linken oder der rechten Seite ein nicht nützliches Nicht-Terminalzeichen haben gestrichen werden, ohne dass sich die erzeugte Sprache ändert. Zur Konstruktion einer Grammatik mit nur nützlichen Nicht-Terminalzeichen werden Regeln mit nicht erreichbaren und nicht erzeugenden Nicht-Terminalzeichen sukzessive eliminiert.

Wir können die erreichbaren Nicht-Terminalzeichen einfach finden, indem wir schrittweise die erreichbaren Nicht-Terminalzeichen markieren. Wir markieren initial das Startsymbol $\underline{S}$. Dann markieren wir schrittweise immer die Nicht-Terminalzeichen jeder Regel $\underline{A} \to w$ mit $w \in (N \cup T)^*$, die links ein markiertes Nicht-Terminalzeichen haben. Das machen wir solange, bis wir kein Nicht-Terminalzeichen mehr neu markieren. Alle noch nicht markierten Nicht-Terminalzeichen sind nicht erreichbar.

Die erzeugenden Nicht-Terminalzeichen können wir auf ähnliche Weise finden. Wir beginnen wieder mit allen Produktion und alle Zeichen unmarkiert. Dann markieren wir zunächst alle Nicht-Terminalzeichen A, die auf der rechten Seite einer Produktion ein Wort aus Terminalzeichen haben. Mit anderen Worten, wir markieren alle A mit $A \to w \in P$ mit $w \in T^*$. Dann markieren wir schrittweise alle Nicht-Terminalzeichen A, die auf der rechten Seite einer Regel *nur* aus Terminalzeichen oder schon markierten Nicht-Terminalzeichen bestehen, da man diese auch in ein $w \in T^*$ überführen kann – wenn auch nur in mehreren Schritten. Das machen wir solange in einem Durchlauf noch Nicht-Terminalzeichen neu markiert werden. Alle schließlich nicht markierten Nicht-Terminalzeichen sind nicht erzeugend.

Wir wiederholen den Prozess solange, bis keine Änderungen mehr eintreten.

Beispiel 4.11. Wir haben folgende fünf Produktionen.

$$\begin{array}{lcl} S & \rightarrow & AB|aA \\ A & \rightarrow & b \\ B & \rightarrow & AB \\ C & \rightarrow & c \end{array}$$

Wir markieren durch Unterstreichen. Zunächst berechnen wir die erreichbaren Nicht-Terminalzeichen. Wir markieren zunächst S. Dann alle Nicht-Terminalzeichen einer Regel, die links ein erreichbares Nicht-Terminalzeichen haben, solange bis sich nichts mehr ändert.

$$\begin{array}{lcl} \underline{S} & \rightarrow & AB|aA \\ A & \rightarrow & b \\ B & \rightarrow & AB \\ C & \rightarrow & c \end{array} \qquad\qquad \begin{array}{lcl} \underline{S} & \rightarrow & \underline{AB}|a\underline{A} \\ \underline{A} & \rightarrow & b \\ \underline{B} & \rightarrow & \underline{AB} \\ C & \rightarrow & c \end{array}$$

Wir haben C nicht markiert, also ist C nicht erreichbar. Wir erhalten

$$\begin{array}{lcl} S & \rightarrow & AB|aA \\ A & \rightarrow & b \\ B & \rightarrow & AB \end{array}$$

Dann berechnen wir die erzeugenden Nicht-Terminalzeichen. Wir markieren initial nur A. In einer zweiten Runde, markieren wir dann S wegen $S \rightarrow a\underline{A}$. Mehr kann nicht markiert werden.

$$\begin{array}{lcl} S & \rightarrow & \underline{A}B|a\underline{A} \\ \underline{A} & \rightarrow & b \\ B & \rightarrow & \underline{A}B \end{array} \qquad\qquad \begin{array}{lcl} \underline{S} & \rightarrow & \underline{A}B|a\underline{A} \\ \underline{A} & \rightarrow & b \\ B & \rightarrow & \underline{A}B \end{array}$$

Also ist B ist nicht erzeugend. Wir eliminieren alle Regeln mit B und erhalten.

$$\begin{array}{lcl} S & \rightarrow & aA \\ A & \rightarrow & b \end{array}$$

Ein weiterer Durchlauf findet nur noch nützliche Nicht-Terminalzeichen. Die Sprache der (ursprünglichen als auch der letzten) Grammatik ist also $\{ab\}$.

Wir gehen im Folgenden davon aus, dass wir nur Grammatiken mit ausschließlich nützlichen Nicht-Terminalzeichen haben.

Definition 4.6. Eine Produktionsregel der Form

$$A \rightarrow B$$

mit $A, B \in N$ heißt Einheitsproduktion.

Satz 4.4. Zu jeder kontextfreien Grammatik G gibt es eine kontextfreie Grammatik G' ohne Einheitsproduktionen mit $L(G') = L(G)$.

Einheitsproduktionen sind auch unerwünscht und sollen eliminiert werden. Wir gehen dazu von einer ε-freien Grammatik aus.

Zunächst überprüfen wir ob es Zyklen gibt. Ein *Zyklus* ist eine Menge von Einheitsproduktionen der Form

$$Z = \{A_1 \to A_2, A_2 \to A_3, \dots, A_{n-1} \to A_n, A_n \to A_1\}$$

mit $A_i \in N$ für $i = 1, \dots, n$. Solange ein Zyklus Z existiert, werden alle Produktionen in Z gelöscht und jedes Vorkommen der Nicht-Terminalzeichen $A_1, \dots, A_n$ durch ein neues Nicht-Terminalzeichen B, das in N aufgenommen wird, ersetzt.

Anschließend ersetzen wir sukzessive jede Einheitsproduktion $A \to B$ durch alle möglichen Ersetzungen von B. Wir löschen also die Produktion $A \to B$ und fügen die Menge

$$\{A \to w \mid B \to w \in P\}$$

zu der Menge von Produktionen P hinzu. Dies machen wir solange bis alle Einheitsproduktionen eliminiert sind.

Wir kommen nun zu den eigentlichen Normalformen.

Definition 4.7. Eine kontextfreie Grammatik heißt in *Greibach-Normalform*, falls nur Regeln der Form $A \to aw$ mit $a \in T$ und $w \in (N \cup T)^*$ auftreten.

Satz 4.5. Zu jeder kontextfreien Grammatik G mit der Sprache $L(G)$ gibt es eine Grammatik G' in Greibach-Normalform mit $L(G') = L(G) - \{\varepsilon\}$.

Mit einer Grammatik in Greibach-Normalform wird bei jedem Schritt in einer Linksableitung mindestens ein neues Nicht-Terminalzeichen rechts angefügt. Der Ableitungsbaum wird also systematisch aufgebaut und eignet sich auf den ersten Blick gut zur algorithmischen Verwendung. Leider ist die Greibach-Normalform nur schwer zu berechnen und die Struktur meist nicht mehr verständlich.

Definition 4.8. Eine kontextfreie Grammatik heißt in *Chomsky-Normalform*, falls nur Regeln der Form $A \to BC$ oder $A \to a$ auftreten und nur nützliche Nicht-Terminalzeichen auftreten.

Satz 4.6. Zu jeder kontextfreien Grammatik G mit der Sprache $L(G)$ gibt es eine Grammatik G' in Chomsky-Normalform mit $L(G') = L(G) - \{\varepsilon\}$.

Um aus einer beliebigen Grammatik eine Grammatik in Chomsky-Normalform (CNF) zu konstruieren, machen wir die Grammatik zunächst ε-frei und stellen dann sicher, dass nur nützliche Nicht-Terminalzeichen und keine Einheitsproduktionen vorhanden sind. Wir können also davon ausgehen, dass alle Produktionen entweder die Form $A \to a$ haben mit

$A \in N$ und $a \in T$ oder die Form $A \to w$ mit $A \in N, w \in (N \cup T)^*$ und $|w| \geq 2$. Die Produktionen der Form $A \to a$ sind in CNF schon zulässig. Es geht also nur noch um die Produktionen $A \to w$ mit rechten Seiten w, die mehr als zwei Zeichen haben.

Um diese Produktionen zu eliminieren, ersetzen wir zunächst jedes Terminalzeichen b in den rechten Seiten einer Regel $A \to w$ mit $|w| \geq 2$ durch ein neues Nicht-Terminalzeichen B und fügen jeweils die Produktion $B \to b$ hinzu. Wir haben dann nur noch Regeln der Form $A \to w$ wobei entweder $|w| = 1, w \in T$ oder $|w| \geq 2, w \in N^*$.

Die Regeln $A \to w$ mit $|w| = n \geq 2$ haben also die Form

$$A \to A_1 \dots A_n$$

wobei die A_i alle Nicht-Terminalzeichen sind. Falls $n = 2$, dann ist die Produktion zulässig in CNF. Falls $n > 2$, dann wird die Produktion ersetzt durch die folgenden $n-1$ Produktionen:

$$A \to A_1B_1 \qquad B_1 \to A_2B_2 \quad B_2 \to A_3B_3 \quad \dots \quad B_{n-3} \to A_{n-2}B_{n-2} \quad B_{n-2} \to A_{n-1}A_n$$

Dabei sind die B_i jeweils neue Nicht-Terminalzeichen. Alle Produktionen sind in CNF zulässig. Dies wird solange durchgeführt, bis nur noch Produktionen übrig sind, die in CNF zulässig sind.

Beispiel 4.12. Korrekte Klammerausdrücke in Chomsky-Normalform.

$$\begin{aligned} T &= \{(,)\} \\ P &= \{S \to SS, \quad S \to (S), \quad S \to \varepsilon\} \end{aligned}$$

Eine Ableitung für den Klammerausdruck $(()(()))$ ist

$$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (()S) \Rightarrow (()(S)) \Rightarrow (()((S))) \Rightarrow (()(()))$$

Wir leiten für Klammerausdrücke eine Grammatik in Chomsky-Normalform her. Zunächst erzeugen wir eine ε-freie Grammatik mit folgender Menge von Produktionen.

$$\{S \to SS, \quad S \to (S) \quad S \to ()\}$$

Beachten Sie, dass $S \Rightarrow^* \varepsilon$ nicht mehr hergeleitet werden kann.

Danach stellen wir sicher, dass nur Nicht-Terminalzeichen in den rechten Seiten mit mehr als zwei Zeichen auftauchen.

$$\{S \to SS, \quad S \to K_aSK_z, \quad S \to K_aK_z, \quad K_a \to (, \quad K_z \to)\}$$

Schließlich konstruieren wir aus der Menge der Produktionen, die nur Nicht-Terminalzeichen auf der rechten Seite haben, eine Menge von Produktionen, die jeweils nur zwei Nicht-Terminalzeichen auf der rechten Seite haben.

$$\{S \to SS, \quad S \to K_aH, \quad H \to SK_z, \quad S \to K_aK_z, \quad K_a \to (, \quad K_z \to)\}$$

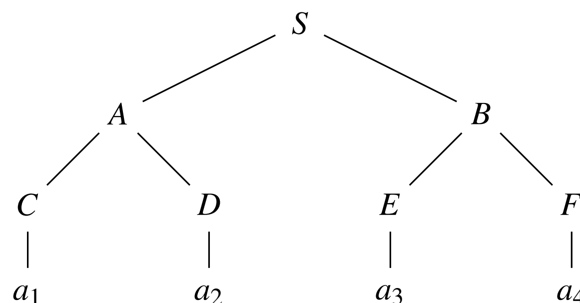
Diese Menge von Produktionen ist dann in Chomsky-Normalform.

Eine Ableitung für $(()(()))$ in CNF ist

$$S \Rightarrow K_a H \Rightarrow^* (SK_z \Rightarrow^* (SS) \Rightarrow^* (K_a K_z K_a H) \Rightarrow^* (()(SK_z) \Rightarrow^* (()(K_a K_z)) \Rightarrow^* (()(()))$$

In der Ableitung wurden in jedem Schritt alle vorhandenen Nicht-Terminalzeichen ersetzt.

Hier ein weiteres Beispiel eines Ableitungsbaums einer Grammatik in der Chomsky-Normalform.



Ableitungsbäume bei der Chomsky-Normalform sind Binärbäume. Wir ignorieren dabei die letzte Ebene und betrachten nur die Ebenen mit Nicht-Terminalzeichen. Im obigen Beispiel ist es also ein Binärbaum der Höhe $n = 3$. Da die Ableitungsbäume Binärbäume sind, können wir Aussagen über die maximale Länge eines erzeugten Wortes in Abhängigkeit zur Höhe des Binärbaums machen.

Satz 4.7. Sei G eine Grammatik in Chomsky-Normalform. Wenn ein Wort $w \in T^*$ aus S abgeleitet wird und der längste Pfad aus Nicht-Terminalzeichen im Ableitungsbaum die Länge n hat, dann ist das Wort w nicht länger als 2^{n-1}.

Beweis. Die Länge n des längsten Pfades aus Nicht-Terminalzeichen im Ableitungsbaum entspricht der Höhe und damit der Anzahl der Nicht-Terminal-Ebenen. Beim Übergang auf die nächste Ebene mit Nicht-Terminalzeichen verdoppelt sich die Zeichenmenge. Beim letzten Übergang auf das Terminalzeichen bleibt die Zeichenmenge gleich. Also kann sich in jeder Ebene die Zeichenmenge höchstens verdoppeln. Es gilt also $|w| \leq 2^{n-1}$. □

Diese Eigenschaft hilft uns allgemeine Aussagen über kontextfreie Sprachen zu machen. Wir können so wieder für alle Worte einer kontextfreien Sprache ab einer gewissen Länge finden neue Worte generieren, indem wir Teilworte wiederholen. Dies kann wieder genutzt werden, um zu zeigen, dass eine Sprache nicht kontextfrei ist.

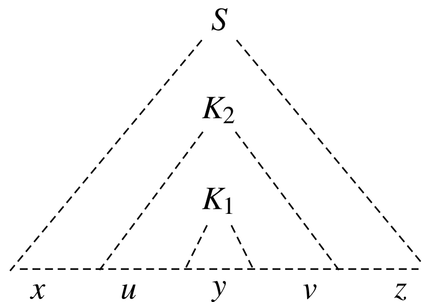
4.3 Das Pumping-Lemma für kontextfreie Sprachen

Satz 4.8. Es sei $L(G)$ die Sprache einer kontextfreien Grammatik $G = (N, T, P, S)$. Dann gibt es eine von G abhängige Zahl k, so dass jedes Wort $w \in L(G)$ mit der Länge $|w| \geq k$ in der Form $w = xuyvz$ mit $x, u, y, v, z \in T^*$ geschrieben werden kann, wobei gilt:

$$\begin{aligned} |uyv| &\leq k \\ uv &\neq \varepsilon \\ xu^iyv^iz &\in L(G) \text{ für } i \geq 0 \end{aligned}$$

Beweis. Wir gehen davon aus, dass G in der Chomsky-Normalform gegeben ist und wählen $k = 2^n$, mit n = Anzahl der Nicht-Terminalzeichen.

Sei für ein Wort w nun $|w| \geq 2^n > 2^{n-1}$, dann muss die Anzahl der Ableitungsebenen mit Nicht-Terminalzeichen größer als n sein, das heißt es gibt im Ableitungsbaum eine Knotenfolge von einem Blatt bis zu der Wurzel mit mindestens $n+1$ Nicht-Terminalzeichen. Da nur n verschiedene existieren, tritt also ein Nicht-Terminalzeichen A mehrfach auf. Wir bezeichnen die letzte Wiederholung mit K_1, die vorletzte mit K_2 und erhalten:



Es gibt $n+1$ Ebenen, die n Nicht-Terminal Ebenen haben. Es ist also y aus K_1 abgeleitet, uyv aus K_2 und $xuyvz$ aus S.

- Für die Ableitung von werden uyv höchstens $n+1$ Schritte benötigt. Es gilt, dass $|uyv| \leq 2^n = k$.
- Da K_2 zwei Nachfolger hat, muss wenigstens aus einem ein nicht leeres Wort u oder v hervorgehen. Also ist $uv \neq \varepsilon$.
- Die Ableitungsschritte des Teilbaums ab K_1 können bereits ab K_2 erfolgen. Die Ableitungsschritte ab K_2 können auch ab K_1 wiederholt werden. Es gilt die Aussage $xu^iyv^iz \in L(G)$ für alle $i \geq 0$.

□

Es könnte mit dem Pumping-Lemma für reguläre Sprachen nachgewiesen werden, dass gewisse Sprachen nicht regulär sind. In ähnlicher Weise kann auch mit dem Pumping-Lemma für kontextfreie Sprachen indirekt nachgewiesen werden, dass gewisse Sprachen nicht kontextfrei sein können. Wir wollen das am nachfolgenden Beispiel demonstrieren.

Satz 4.9. Die Sprache $L = \{a^m b^m c^m | m = 1, 2, \ldots\}$ ist nicht kontextfrei.

Beweis. Angenommen L wäre kontextfrei, dann gilt das Pumping-Lemma (Satz 4.8) für eine bestimmte Zahl k. Wir wählen das Wort $w = a^k b^k c^k$. Offensichtlich gilt $w \in L$. Laut Pumping-Lemma gibt es eine Zerlegung von w in $xuyvz$ für die gilt:

$$\begin{aligned} |uyv| &\leq k \\ uv &\neq \varepsilon \\ xu^i yv^i z &\in L \text{ für } i \geq 0 \end{aligned}$$

Da $|uyv| \leq k$ wissen wir, dass nicht sowohl a's als auch c's in uyv enthalten sein können, da mindestens k mal das b zwischen den a's und den c's liegt. Laut Pumping-Lemma ist für alle $i = 0, 1, 2, \ldots$ das Wort $xu^i yv^i z$ auch in der Sprache. Wir wählen $i = 0$ und schließen, dass auch xyz in L ist. Wir unterscheiden zwischen zwei Fällen:

1. uyv enthält keine c's: Dann besteht auch uv ausschließlich aus a's und b's. uv beinhaltet mindestens ein Zeichen. Damit hat xyz entweder weniger a's oder weniger b's (oder beides) als c's, da mindestens entweder ein a oder ein b gegenüber $xuyvz$ fehlt. Damit wäre xyz nicht in L was zu einem Widerspruch führt.

2. uyv enthält keine a's: Dann besteht auch uv aus ausschließlich aus b's oder c's. uv beinhaltet mindestens ein Zeichen. Damit hat xyz entweder weniger c's oder weniger b's (oder beides) als a's, da mindestens entweder ein b oder ein c gegenüber $xuyvz$ fehlt. Damit wäre xyz nicht in L was zu einem Widerspruch führt.

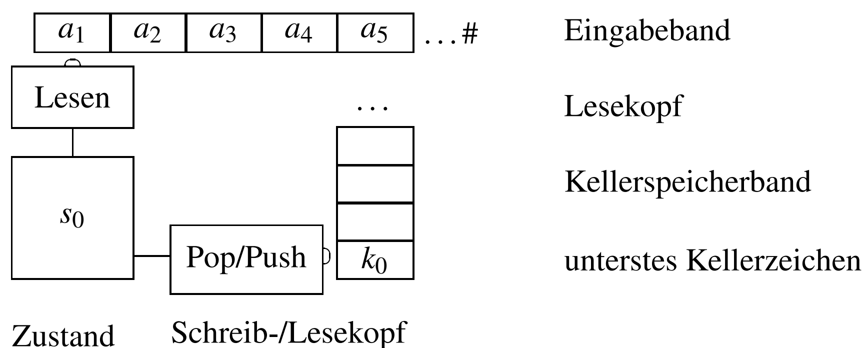
Egal welcher Fall gilt, wir bekommen einen Widerspruch. Also ist unsere Annahme falsch. L ist also nicht kontextfrei. □

Kapitel 5

Kellerautomaten und kontextfreie Sprachen

5.1 Allgemeines zu Kellerautomaten

Eine Erweiterung des endlichen Automaten ist der *Kellerautomat* (KA) oder Pushdown-Automat. Die Arbeitsweise eines Kellerautomaten kann man sich, wie in Abschnitt 1.2 beschrieben, ähnlich wie zu dem eines endlichen Automaten vorstellen. Es besitzt zusätzlich noch ein Kellerspeicherband, auf dem mit einem Schreib-/Lesekopf gearbeitet wird.



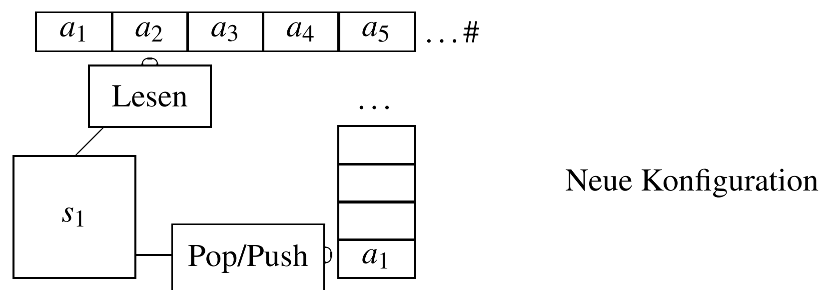
Die graphische Darstellung stellt die „Ausgangskonfiguration“ dar. Der Kellerautomat ist im inneren Zustand s_0, der Lesekopf steht auf dem 1. Zeichen a_1 eines Wortes $a_1a_2a_3\ldots$. Der Schreib-/Lesekopf des Kellerbandes steht auf dem initialen speziellen Kellerzeichen k_0, das den ansonsten leeren Keller signalisiert. Mögliche Aktionen sind:

- ein Zeichen vom Eingabeband lesen
- ein Zeichen vom Kellerband lesen und entfernen (Pop)
- mehrere Zeichen auf das Kellerband schreiben (Push)
- interner Zustandswechsel

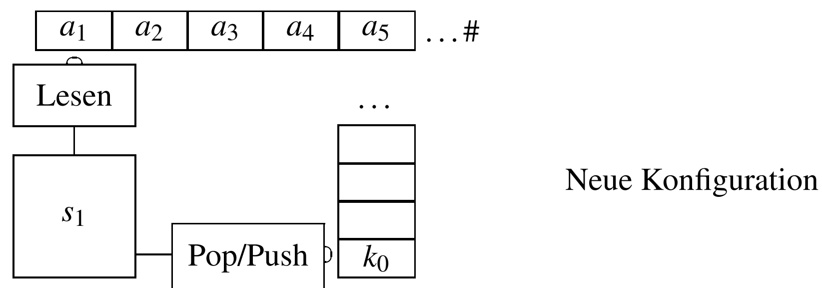
Die möglichen Aktionen werden formal wieder durch eine Überführungsfunktion beschrieben, die außer von einem inneren Zustand und einem Eingabezeichen noch von dem jeweiligen obersten Kellerzeichen abhängt.

Nachfolgend die Überführungsfunktion und Arbeitsschritte anhand von Beispielen.

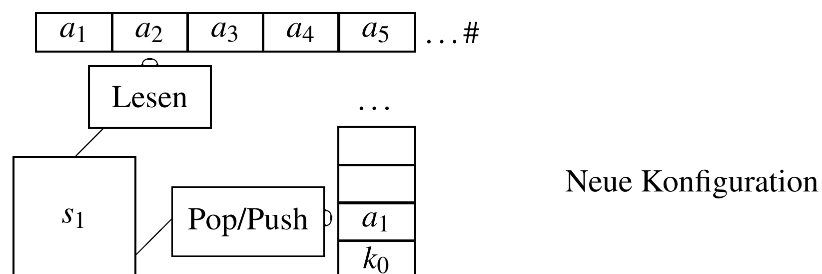
1. $\delta(s_0, a_1, k_0) = (s_1, a_1)$: Lies a_1, Pop k_0, neuer Zustand ist s_1, Push a_1 (anstelle k_0) ergibt folgende neue Konfiguration.



2. $\delta(s_0, \varepsilon, k_0) = (s_1, k_0)$: Lies nicht(!), Pop k_0, neuer Zustand ist s_1, Push k_0



3. $\delta(s_0, a_1, k_0) = (s_1, a_1 k_0)$: Lies a_1, Pop k_0, neuer Zustand ist s_1, Push $a_1 k_0$



Eine Kurzschreibweise für Konfigurationen ist ein Tupel mit drei Elementen: Ausgangszustand, noch einzulesendes Wort und aktuelles Kellerband. Die Kurzschreibweise für die Ausgangskonfiguration wäre also $(s_0, a_1a_2a_3\ldots\#, k_0)$. Nachfolgend finden Sie obige Beispiele in der Kurzschreibweise.

Ausgangskonfiguration	Regel	Zielkonfiguration
$(s_0, a_1a_2a_3a_4a_5\#, k_0)$	$\delta(s_0, a_1, k_0) = (s_1, a_1)$	$(s_1, a_2a_3a_4a_5\#, a_1)$
	$\delta(s_0, \varepsilon, k_0) = (s_1, k_0)$	$(s_1, a_1a_2a_3a_4a_5\#, k_0)$
	$\delta(s_0, a_1, k_0) = (s_1, a_1k_0)$	$(s_1, a_2a_3a_4a_5\#, a_1k_0)$
$(s_1, a_2a_3a_4a_5\#, a_1k_0)$	$\delta(s_1, a_2, a_1) = (s_2, \varepsilon)$	$(s_2, a_3a_4a_5\#, k_0)$

Wenn wir von einer Konfiguration (s, w, φ) in einem Schritt (mit Anwendung einer Regel) in eine Konfiguration (s', w', φ') kommen, dann schreiben wir auch

$$(s, w, \varphi) \vdash (s', w', \varphi') \ .$$

Wenn wir von einer Konfiguration (s, w, φ) in keinem, einem oder mehreren Schritten in eine Konfiguration (s', w', φ') kommen, dann schreiben wir auch

$$(s, w, \varphi) \vdash^* (s', w', \varphi') \ .$$

5.2 Deterministische Kellerautomaten

Definition 5.1. Das Tupel $A = (S, s_0, F, \Sigma, K, k_0, \delta)$ ist ein *deterministischer Kellerautomat* (DKA), wenn die einzelnen Komponenten Folgendes bedeuten:

S Endliche Menge der internen Zustände des Automaten

s_0 Interner Anfangszustand des Automaten, $s_0 \in S$

F Menge der internen Endzustände des Automaten, $F \subseteq S$

Σ Endliche Menge der Eingabezeichen

K Endliche Menge der Kellerzeichen k

k_0 Kellerstartzeichen (unterstes Zeichen auf dem Kellerband)

δ Überführungsfunktion $S \times (\Sigma \cup \{\varepsilon\}) \times K \to S \times K^*$, wobei für ein Paar $(s, k) \in S \times K$ entweder $\delta(s, \varepsilon, k)$ oder $\delta(s, a, k)$ definiert sein darf (ansonsten wäre der Automat nicht deterministisch).

Wir betrachten die Bearbeitung eines Wortes auf dem Eingabeband. Zu Beginn gilt:

- $s = s_0$ ist der aktuelle interne Zustand.

- $a = a_1$ (oder $a = \#$) ist das aktuelle Bandzeichen.
- $k = k_0$ ist das aktuelle Kellerzeichen.

Die Bearbeitung kann mit folgendem Algorithmus beschrieben werden.

```
1  keller = [k0]
2  s = s0
3  i = 1
4  while i <= n:
5      k = keller.top()  # oberstes Kellerzeichen
6      if δ(s,ε,k): # Ueberfuehrung ohne Zeichen
7          (s',k') = δ(s,ε,k)
8      elif δ(s,a_i,k): # Ueberfuehrung mit Zeichen
9          (s',k') = δ(s,a_i,k)
10         i += 1
11     else:  # Ueberfuehrung nicht mehr definiert
12         break # Ende der Verarbeitung
13     keller.pop()
14     keller.push(k') # (mehrere) Zeichen auf Keller
15     s = s'
```

Der DKA A stoppt, wenn die Überführungsfunktion für die aktuelle Konfiguration nicht definiert ist.

Definition 5.2. Der DKA A *akzeptiert* ein Wort w wenn

$$(s_0, w\#, k_0) \vdash^* (s_f, \#, \varphi) \ ,$$

das heißt wenn der DKA von der Ausgangskonfiguration $(s_0, w\#, k_0)$ mit einer Endkonfiguration $(s_f, \#, \varphi)$ stoppt. Dabei muss s_f ein Endzustand sein und φ kann ein beliebiger Kellerinhalt sein kann.

Häufig ist es möglich, mit $\varphi = k_0$ den ursprünglichen Keller wieder herzustellen.

Definition 5.3. Die *Sprache $L(A)$ eines deterministischen Kellerautomaten A* besteht aus allen Worten $w \in \Sigma^*$, die vom DKA akzeptiert werden, das heißt bei denen $(s_0, w\#, k_0) \vdash^* (s_f, \#, \varphi)$ gilt und $(s_f, \#, \varphi)$ mit $s_f \in F$ und $\varphi \in K^*$ die Endkonfiguration ist.

Beispiel 5.1. Ein deterministischer Kellerautomat für $a^n b^n$.

$$\begin{array}{lcl}
\Sigma & = & \{a,b\} \\
S & = & \{s_0,s_1,s_2,s_3\} \\
F & = & \{s_0,s_3\} \\
K & = & \{k_0,a\} \\
\delta & : & \delta(s_0,a,k_0) = (s_1,ak_0), \quad \delta(s_1,a,a) = (s_1,aa), \\
 & & \delta(s_1,b,a) = (s_2,\varepsilon), \quad \delta(s_2,b,a) = (s_2,\varepsilon) \\
 & & \delta(s_2,\varepsilon,k_0) = (s_3,k_0)
\end{array}$$

Wir erhalten folgende Konfigurationsfolge, wenn a^3b^3 auf dem Band steht:

$$\begin{array}{lcll}
(s_0, aaabbb\#, k_0) & \vdash & (s_1, aabbb\#, ak_0) & \vdash \\
(s_1, abbb\#, aak_0) & \vdash & (s_1, bbb\#, aaak_0) & \vdash \\
(s_2, bb\#, aak_0) & \vdash & (s_2, b\#, ak_0) & \vdash \\
(s_2, \#, k_0) & \vdash & (s_3, \#, k_0) &
\end{array}$$

5.3 Nicht-Deterministische Kellerautomaten

Definition 5.4. Das Tupel $A = (S, s_0, F, \Sigma, K, k_0, \delta)$ ist ein *nicht-deterministischer Keller-Automat* (NKA), wenn die ersten Komponenten dieselbe Bedeutung haben wie beim deterministischen Kellerautomaten, aber δ stattdessen eine *nicht-deterministische* Überführungsfunktion von $S \times (\Sigma \cup \{\varepsilon\}) \times K$ in die Potenzmenge von $S \times K^*$ darstellt.

Das bedeutet, dass es Konfigurationen eines nicht-deterministischen Kellerautomaten gibt, für die es mehrere Nachfolgekonfigurationen geben kann. Auch kann beim NKA für einen Zustand s sowohl $\delta(s, a, k)$ als auch $\delta(s, \varepsilon, k)$ definiert sein, das heißt es kann sowohl ein Zeichen gelesen werden als auch spontan in einen anderen Zustand übergegangen werden.

Definition 5.5. Die *Sprache* eines *nicht-deterministischen Kellerautomaten A* wird bezeichnet mit $L(A)$ und besteht aus allen Worten aus Σ^*, bei denen es *möglich* ist, dass der NKA nach endlichen vielen Schritten in einer Endkonfiguration $(s_f, \#, \varphi)$ mit $s_f \in F$ und $\varphi \in K^*$ stoppt.

Beispiel 5.2. Ein nicht-deterministischer Kellerautomat für die Sprache ww^R, wobei w^R die Spiegelung eines Wortes $w \in \Sigma^*$ darstellen soll.

$$\begin{array}{rcl}
\Sigma & = & \{a, b\} \\
S & = & \{s_0, s_1, s_2\} \\
F & = & \{s_2\} \\
K & = & \{k_0, a, b\} \\
\delta & : & \begin{array}{rclrcl}
\delta(s_0, a, k_0) & = & (s_0, ak_0), & \delta(s_0, b, k_0) & = & (s_0, bk_0), \\
\delta(s_0, a, b) & = & (s_0, ab), & \delta(s_0, b, a) & = & (s_0, ba) \\
\delta(s_0, a, a) & = & (s_0, aa), & \delta(s_0, b, b) & = & (s_0, bb) \\
\delta(s_0, \varepsilon, a) & = & (s_1, a), & \delta(s_0, \varepsilon, b) & = & (s_1, b) \\
\delta(s_0, \varepsilon, k_0) & = & (s_2, k_0) & & & \\
\delta(s_1, a, a) & = & (s_1, \varepsilon), & \delta(s_1, b, b) & = & (s_1, \varepsilon) \\
\delta(s_1, \varepsilon, k_0) & = & (s_2, k_0) & & &
\end{array}
\end{array}$$

Satz 5.1. Es gibt keinen *deterministischen* Kellerautomaten, der die Sprache der gespiegelten Worte ww^R akzeptiert.

Zwischen nicht-deterministischen Kellerautomaten und deterministischen besteht also *keine* Äquivalenz wie bei endlichen Automaten. Die Menge der Sprachen, die von NKAs akzeptiert werden ist größer als die, die von DKAs akzeptiert werden.

5.4 Kellerautomaten und kontextfreie Grammatiken

Satz 5.2. Zu jeder kontextfreien Grammatik G gibt es einen nicht-deterministischen Kellerautomat A mit $L(\mathrm{A}) = L(G)$ und umgekehrt.

Beweis. Beweis nur für eine Richtung. Bei gegebener Grammatik $G = (N, T, P, S)$ wird ein nicht-deterministischer Kellerautomat $A = (S, s_0, F, \Sigma, K, k_0, \delta)$ konstruiert.

Zustände $S = \{s_0, s_1, s_f\}$
Endzustände $F = \{s_f\}$
Eingabezeichen $\Sigma = T$
Kellerzeichen $K = \{k_0\} \cup T \cup N$
Überführungsfunktion:

$\delta(s_0, \varepsilon, k_0) = (s_1, Sk_0)$	wobei S das Startsymbol von G ist
$\delta(s_1, \varepsilon, A) = (s_1, \gamma)$	für jede Regel $A \to \gamma$ von G (oberstes Kellerzeichen ist erstes Zeichen von γ)
$\delta(s_1, a, a) = (s_1, \varepsilon)$	für alle Eingabezeichen bzw. Terminalzeichen a
$\delta(s_1, \varepsilon, k_0) = (s_f, k_0)$	(wenn Keller wieder initial ist wird akzeptiert)

Die Überführungsfunktion ermöglicht es, genau die Produktionsregeln, mit denen ein Wort w der Sprache $L(G)$ erzeugt wurde, so auf das Kellerband zu schreiben, dass w als Wort auf dem Eingabeband mit den Funktionswerten $\delta(s_1, a, a) = (s_1, \varepsilon)$ abgearbeitet werden kann. □

Beispiel 5.3. Es sei eine Grammatik G gegeben mit $N = \{S\}, T = \{a, b\}$, den Produktionen $P = \{S \to aSb|\varepsilon\}$ und dem Startsymbol S. Ein Kellerautomat, der die gleiche Sprache akzeptiert die G erzeugt, hat folgende Überführungsfunktion.

$$\begin{aligned}
\delta(s_0, \varepsilon, k_0) &= (s_1, Sk_0) \\
\delta(s_1, \varepsilon, S) &= \{(s_1, aSb), (s_1, \varepsilon)\} \\
\delta(s_1, a, a) &= (s_1, \varepsilon) \\
\delta(s_1, b, b) &= (s_1, \varepsilon) \\
\delta(s_1, \varepsilon, k_0) &= (s_f, k_0)
\end{aligned}$$

Wir geben die Konfigurationsfolge an, die $aabb \in L(G)$ akzeptiert.

$$\begin{array}{lllllll}
(s_0, aabb\#, k_0) & \vdash & (s_1, aabb\#, Sk_0) & \vdash & (s_1, aabb\#, aSbk_0) & \vdash & \\
(s_1, abb\#, Sbk_0) & \vdash & (s_1, abb\#, aSbbk_0) & \vdash & (s_1, bb\#, Sbbk_0) & \vdash & \\
(s_1, bb\#, bbk_0) & \vdash & (s_1, b\#, bk_0) & \vdash & (s_1, \#, k_0) & \vdash & (s_f, \#, k_0)
\end{array}$$

5.5 Endliche Automaten und Kellerautomaten

Intuitiv einleuchtend ist es einen endlichen Automaten durch einen Kellerautomaten zu simulieren. Dazu werden die gleichen inneren Zustände, die gleichen Eingabezeichen und

hinsichtlich dieser beiden Komponenten auch die gleichen Definitionen der Überführungsfunktion benutzt. Den Keller muss die Überführungsfunktion nicht verändern. Das anfängliche Kellerzeichen k_0, bleibt in jedem Arbeitsschritt erhalten.

Beispiel 5.4. Ein endlicher Automat sei definiert durch $\Sigma = \{a, b\}$, $S = \{s_0, s_1, s_2\}$, $F = \{s_2\}$, und Startzustand s_0 sowie nachfolgender Überführungsfunktion.

Überführungsfunktion Endlicher Automat	Überführungsfunktion Kellerautomat
$\delta(s_0, a) = s_1$	$\delta(s_0, a, k_0) = (s_1, k_0)$
$\delta(s_1, a) = s_1$	$\delta(s_1, a, k_0) = (s_1, k_0)$
$\delta(s_1, b) = s_2$	$\delta(s_1, b, k_0) = (s_2, k_0)$
$\delta(s_2, b) = s_2$	$\delta(s_2, b, k_0) = (s_2, k_0)$
$\delta(s_2, a) = s_1$	$\delta(s_2, a, k_0) = (s_1, k_0)$

Die Überführungsfunktion für den entsprechenden Kellerautomaten ist in der rechten Spalte. Der endliche Automat (und der Kellerautomat) erkennt die reguläre Sprache $a(a|b)^*b$, das heißt Worte die mit a beginnen und mit b enden.

Offensichtlich wird bei obiger Simulation aus einem deterministischen endlichen Automat ein deterministischer Kellerautomat und aus einem nicht-deterministischen endlichen Automat ein nicht-deterministischer Kellerautomat.

5.6 Alternativen für das Akzeptieren eines Wortes

Wir haben definiert, dass ein Wort von einem Kellerautomaten akzeptiert wird, wenn der Kellerautomat in einer Endkonfiguration $(s_f, \#, \varphi)$ stoppt mit $s_f \in F$ und $\varphi \in K^*$ beliebig. Eine andere Möglichkeit ist das *Akzeptieren durch leeren Keller*.

Definition 5.6. (Alternative zu Def. 5.2) Ein Kellerautomat akzeptiert ein Wort w wenn

$$(s_0, w\#, k_0) \vdash^* (s, \#, \varepsilon)$$

Dabei muss in der Endkonfiguration s kein Endzustand mehr sein, aber der Keller muss vollständig geleert sein.

Die beiden Definitionen sind insofern äquivalent, als es zu jeder kontextfreien Sprache L sowohl einen Kellerautomaten gibt, der mit Endkonfiguration $(s_f, \#, \varphi)$ als auch mit Endkonfiguration $(s, \#, \varepsilon)$ akzeptiert. Ist der Kellerautomat deterministisch und akzeptiert mit $(s, \#, \varepsilon)$, dann gibt es auch einen äquivalenten deterministischen Kellerautomaten der mit $(s_f, \#, \varphi)$ akzeptiert. Das Umgekehrte gilt jedoch nicht. Der deterministische Kellerautomat, der mit Endzustand akzeptiert kann mehr Sprachen erkennen als der deterministische Kellerautomat, der mit leerem Keller akzeptiert.

Für eine Sprache L, in der es Worte $w = xy$ gibt, deren Präfix x auch schon ein Wort der Sprache ist, kann es keinen deterministischen Kellerautomat geben, der mit leerem Keller L akzeptiert. Dann wäre nämlich der Keller nach dem Lesen des Präfix leer. Aber dann stoppt der Automat und wird ein folgendes y nicht akzeptieren. Wenn der Automat jedoch xy akzeptiert, dann wird der Keller bei x nicht leer sein und kann deswegen x nicht akzeptieren.

Beispiel 5.5. Sei L die Sprache aus Beispiel 5.4, die beschrieben werden kann durch $a(a|b)^*b$. Dann sind sowohl *aabab* als auch *aab* Worte der Sprache L. Ein deterministischer Kellerautomat, der mit leerem Keller akzeptiert muss also nach Lesen von *aab* einen leeren Keller haben. Dieser deterministische Kellerautomat kann dann aber *aabab* nicht mehr erkennen.

Die Sprache aus Beispiel 5.4 ist regulär. Deterministische Kellerautomaten, die mit leerem Keller akzeptieren können noch nicht einmal reguläre Sprachen erkennen.

Man kann dieses Problem beheben, indem man fordert, dass jedes Wort der Sprache mit einem neuen Symbol endet, das bisher nicht in dem Alphabet vorkommt. Dieses Symbol kann zum Beispiel $ sein. Die Worte für Beispiel 5.4 sind dann *aab*$ und *aabab*$. Jetzt ist *aab*$ kein Präfix mehr von *aabab*$. Beim Lesen des neuen Symbols wird dann der Keller geleert.

Kapitel 6

Das Problem der Syntaxanalyse

Höhere Programmiersprachen und Auszeichnungssprachen basieren zum großen Teil auf kontextfreien Grammatiken. Die Frage, ob ein Programm oder Dokument syntaktisch korrekt ist, entspricht der Aufgabe einen Automaten (Algorithmus) zu finden, der genau die Worte der Grammatik akzeptiert.

Aus dem letzten Kapitel wissen wir, dass es zu einer kontextfreien Grammatiken G zwar immer einen *nicht-deterministischen* Kellerautomaten A mit $L(G) = L(A)$ gibt, aber nicht immer einen *deterministischen* Kellerautomaten. Infolgedessen ist mit Hilfe von Kellerautomaten auch kein effizientes Verfahren zur Syntaxanalyse herleitbar, das für alle kontextfreien Grammatiken genutzt werden kann. Man muss entweder ein Verfahren finden, das nicht auf einem Kellerautomaten basiert oder man muss sich auf kontextfreie Grammatiken beschränken, für die es deterministische Kellerautomaten gibt. Auf beides wollen wir kurz eingehen.

6.1 Das CYK-Verfahren

Ein naives Verfahren um festzustellen, ob ein Wort in der Sprache einer kontextfreien Grammatik ist hat im schlimmsten Fall eine exponentielle Laufzeit. Ein effizienteres Verfahren wurde von Cocke, Younger und Kasami (CYK) entwickelt.

Eine Grammatik $G = (N, T, P, S)$ sei in der Chomsky-Normalform gegeben, das heißt es gibt nur Produktionen der Form $A \to a$ oder $A \to BC$. Sei $w = a_1 a_2 \dots a_n$ zu untersuchen:

- Betrachte die Teilworte $a_i a_{i+1} \dots a_j$ für $1 \leq i \leq j \leq n$ und die zugehörigen Mengen

$$M(i, j) = \{A \in N | A \Rightarrow^* a_i a_{i+1} \dots a_j\} \ .$$

- Falls S in der Menge $M(1, n)$, dann gilt $S \Rightarrow^* a_1 a_2 \dots a_n = w$ und damit ist w in $L(G)$. Die Umkehrung gilt auch. Daher gilt

$$w \in L(G) \ \text{gdw} \ S \in M(1, n) \ .$$

Die Mengen $M(i,j)$ können schrittweise wie folgt konstruiert werden.

```
1 for i ∈ {1,2,...,n}: # Initialisierung i = j
2     M(i,i) = {A|A → a_i ∈ P}
3 for s ∈ {1,...,n−1}:
4     for i ∈ {1,...,n−s}:
5         M(i,i+s) = {A | A → BC ∈ P,
                          B ∈ M(i,k),
                          C ∈ M(k+1,i+s),
                          k = i,...,i+s−1}
```

Das Verfahren kann mit Hilfe einer Dreieckstabelle, deren Felder für die Aufnahme der Mengen $M(i,j),(i \leq j = 1,2,\ldots,n)$ dienen, gut illustriert werden, wie man an folgendem Beispiel sieht.

Beispiel 6.1. Die Sprache $L = \{a^n b^n c^m \mid n,m \geq 1\}$ sei durch folgende Produktionen einer Grammatik definiert:

$$S \to DE, \quad D \to aDb|ab, \quad E \to cE|c$$

Diese Produktionen in Chomsky-Normalform überführt sind:

$$\begin{array}{llll} S \to DE, & D \to HB|AB, & H \to AD, & E \to CE, \\ A \to a, & B \to b, & C \to c, & E \to c \end{array}$$

Untersucht werden soll das Wort $aabbcc = a_1a_2a_3a_4a_5a_6$.

	1	2	3	4	5	6
1	$_a$ A	.	H	D	S	S
2		$_a$ A	D	.	.	.
3			$_b$ B	.	.	.
4				$_b$ B	.	.
5					$_c$ C,E	E
6						$_c$ C,E

Das Startsymbol S ist in $M(1,6)$. Daher ist $aabbcc$ in der Sprache.

Man kann mit dem CYK-Algorithmus entscheiden, ob ein Wort w in der Sprache einer beliebigen kontextfreien Grammatik enthalten ist. Die Komplexität dieses Verfahrens ist $O(n^3)$.

Die Komplexität des CYK-Algorithmus ist für viele praktische Anwendungen immer noch nicht ausreichend. Für den Bau von Compilern wäre eine Syntaxanalyse mit der Komplexität $O(n)$ wünschenswert. Dies ist für eine Untermenge der kontextfreien Sprachen, nämlich für diejenigen, für die eine sogenannte LL(k)- oder LR(k)-Grammatik existiert, machbar.

6.2 Syntaxanalyse mit LL(k)-Grammatiken

Bei der Syntaxanalyse mit LL(k)- (und LR(k))-Grammatiken erfolgt die Untersuchung eines Wortes grundsätzlich *von links nach rechts*. Dabei wird maximal k Zeichen nach vorne geschaut, um eindeutig festzulegen welche Regel angewendet werden muss, um das untersuchte Wort zu erzeugen. Die Untersuchungsrichtung von **L**inks nach rechts steht für das erste Zeichen L in LL(k)- und LR(k)- Grammatiken. Das k für die Anzahl der vor der eindeutigen Entscheidung zu untersuchenden Zeichen hinter der aktuellen Position. Das zweite Zeichen bedeutet, dass entweder eine **L**inksableitung oder eine **R**echtsableitung konstruiert wird.

Beim Top-Down-Verfahren wird versucht den Syntaxbaum vom Startsymbol S aus (Top) zu den Blättern hin (Down) aufzubauen. In Verbindung mit der Analyserichtung von links nach rechts ist das gleichwertig damit, die *Linksableitung* des zu analysierenden Wortes zu rekonstruieren. Dieses Vorgehen wird bei der LL(k)-Grammatik eingesetzt. Beim Bottom-Up-Verfahren wird versucht den Syntaxbaum von den Blättern (Bottom) her zum Startsymbol hin (Up) aufzubauen. Das geschieht durch sogenannte Reduktionsschritte, bei denen in der jeweiligen Satzform nach der rechten Seite einer Regel gesucht wird, um sie durch die linke Seite zu ersetzen. In Verbindung mit der Analyserichtung von links nach rechts ist das gleichwertig damit, die *Rechtsableitung* des zu analysierenden Wortes zu rekonstruieren. Dieses Vorgehen wird bei der LR(k)-Grammatik eingesetzt. Wir beschränken uns im Folgenden auf LL(k)-Grammatiken.

Beim Bearbeiten des vorgegebenen Worts von Links nach rechts kann mit einem *lookahead* von k Zeichen die Linksableitung rekonstruiert werden. Beachten Sie, dass *lookahead 1* bedeutet, das aktuelle (unterstrichene) Zeichen zu lesen und *lookahead 2* auch zusätzlich das nachfolgende Zeichen zu berücksichtigen.

Das Verfahren kann mit einem deterministischen Kellerautomaten ausgeführt werden, bei dem im Wesentlichen die erkannte Regel in den Keller geschrieben und das oberste Zeichen (erstes Zeichen der Regel) mit dem aktuellen Eingabezeichen abgearbeitet wird.

Beispiel 6.2. Wir machen eine Top-Down-Analyse mit einer LL(2)-Grammatik. Mit den Produktionen

$$S \to aSB|aB, \quad B \to b|\varepsilon$$

einer Grammatik G erzeugen wir die Sprache $L(G) = \{a^n b^m | m \leq n = 1, 2, \ldots\}$. Wir wollen das Wort $w = aaabb$ untersuchen.

Position	anzuwendende Regel	lookahead	erzeugte Satzform
$\underline{a}aabb\$$	$S \to aSB$	2	aSB
$a\underline{a}abb\$$	$S \to aSB$	2	$aaSBB$
$aa\underline{a}bb\$$	$S \to aB$	2	$aaaBBB$
$aaa\underline{b}b\$$	$B \to b$	1	$aaabBB$
$aaab\underline{b}\$$	$B \to b$	1	$aaabbB$
$aaabb\underline{\$}$	$B \to \varepsilon$	1	$aaabb$

Es ist auch möglich, das Verfahren direkt mit einem geeignet konstruierten Kellerautomaten auszuführen in dem die lookahead-Operationen in die Ausführung eingebunden sind.

Satz 6.1. Für jede LL(k)-Grammatik G kann man einen deterministischen Kellerautomaten A angeben, so dass

$$w \in L(G) \quad \Leftrightarrow \quad w\$ \in L(A) \ .$$

Beweis. Wir konstruieren für den Beweis einen deterministischen Kellerautomaten $A = (S, s_0, F, \Sigma, K, k_0, \delta)$ aus einer LL(k)-Grammatik $G = (N, T, P, S)$. Die Konstruktion beruht auf derselben Idee wie das Verfahren in Satz 5.2 – Ausführen der Ersetzungsregeln auf dem Keller – wobei die Überführungsfunktion jetzt deterministisch ist.

In den Zuständen des Kellerautomaten werden die gelesenen aber noch nicht verarbeiteten Zeichen gespeichert. Bei einer LL(k)-Grammatik müssen wir dazu maximal k Zeichen speichern. Diese maximal k gespeicherten Zeichen bestimmen eindeutig die anzuwendende Regel. Wir müssen also zusätzlich Zustände für jedes Wort v mit bis zu k Zeichen vorsehen. Wir nennen diese Menge von Zuständen $S_k = \{s_v \mid v \in \Sigma^*, |v| \leq k\}$.

Zustände	$S = \{s_0, s_f\} \cup S_k$
Startzustand	s_0
Endzustände	$F = \{s_f\}$
Eingabezeichen	$\Sigma = T \cup \{\$\}$
Kellerzeichen	$K = N \cup T$

Für die Überführungsfunktion übernehmen wir die Definition aus Satz 5.2 für alle Terminalzeichen, die Initialisierung und das Akzeptieren. Dabei werden Nicht-Terminalzeichen oben auf dem Stack zuerst gegen gemerkte Zeichen ausgelöscht. Falls keine Zeichen gemerkt sind, dann wird gegen das Eingabewort gelöscht.

$\delta(s_0, \varepsilon, k_0)$	$=$	(s_ε, Sk_0)	wobei S das Startsymbol von G ist
$\delta(s_{av}, \varepsilon, a)$	$=$	(s_v, ε)	für alle $a \in T$ und $s_{av} \in S_k$
$\delta(s_\varepsilon, a, a)$	$=$	$(s_\varepsilon, \varepsilon)$	für alle $a \in T$
$\delta(s_\$, \varepsilon, k_0)$	$=$	(s_f, k_0)	akzeptiert bei Keller k_0, letztes Zeichen gelesen

Für die Behandlung der Nicht-Terminalzeichen nutzen wir aus, dass die Grammatik eine LL(k)-Grammatik ist. Jedes Nicht-Terminalzeichen A hat eine bestimmte Anzahl an Alternativen $\gamma_1, \dots \gamma_n$.

$$A \to \gamma_1 | \cdots | \gamma_n$$

Da G eine LL(k)-Grammatik ist, gibt es für jedes Nicht-Terminalzeichen ein Wort x mit maximaler Länge k, das eindeutig entscheidet welche rechte Seite einer Regel anzuwenden ist. Aus der Analyse der Grammatik können wir also eine Funktion

$$\mathsf{select}(x, A) = \gamma_i \qquad \text{für ein i mit } 1 \leq i \leq n$$

für jedes Nicht-Terminalzeichen A erhalten, die für mindestens ein $x \in \Sigma^*$ mit $|x| \leq k$ für jedes γ_i definiert ist. Wir wählen jeweils das kürzeste x. Wir definieren nun die Überführungsfunktion δ so, dass

- wenn genügend Zeichen gespeichert wurden, die passende Regel ausgeführt wird.
- wenn nicht genügend Zeichen gespeichert wurden, ein weiteres Eingabezeichen gespeichert wird.

$$
\begin{array}{lcll}
\delta(s_{xy},\varepsilon,A) & = & (s_{xy},\gamma_i) & \text{für jede Regel } A \to \gamma_1|\cdots|\gamma_n \text{ von } G \\
 & & & \text{wenn } \mathsf{select}(x,A) = \gamma_i \text{ definiert für ein Präfix } x \\
\delta(s_{xy},a,A) & = & (s_{xya},A) & \text{für jede Regel } A \to \gamma_1|\cdots|\gamma_n \text{ von } G \\
 & & & \text{wenn } \mathsf{select}(x,A) \text{ nicht definiert für alle Präfixe } x
\end{array}
$$

Die Überführungsfunktion ist deterministisch, da für jeden Zustand nur jeweils die eine oder die andere Definition angewendet wird.

Da G eine LL(k)-Grammatik ist, müssen wir nur maximal k Zeichen in x speichern bis $\mathsf{select}(x,A)$ definiert ist. Wir kommen also mit den Zuständen aus S_k aus und können immer eine Regel anwenden, wenn das Eingabewort in der Sprache ist.

Beachten Sie, dass es möglich ist auf einige Zustände in S_k zu verzichten. Es werden nur die erreichbaren Zustände gebraucht. Dies sind die Zustände $s_v \in S_k$ bei denen v ein Präfix ist von x für jedes x und A für die $\mathsf{select}(x,A)$ definiert ist.

Aus der Konstruktion sehen wir, dass der Kellerautomat nur Worte $w\$$ akzeptiert für die es eine Linksableitung $S \Rightarrow_G^* w$ gibt.

□

Beispiel 6.3. Sei die Grammatik $G = (\{S,B\},\{a,b\},P,S)$ gegeben mit den Produktionen

$$P = \{S \to aSB|aB, \quad B \to b|\varepsilon\} \ .$$

Dabei ist $L(G) = \{a^n b^m | m \leq n, n = 1,2,\ldots\}$. Die Grammatik G ist eine LL(2)-Grammatik, das heißt es müssen maximal zwei Zeichen eingelesen werden. Wir konstruieren den Kellerautomaten.

$$
\begin{array}{rcl}
S & = & \{s_0, s_f\} \cup S_k \\
S_k & = & \{s_\varepsilon, s_a, s_b, s_\$, s_{aa}, s_{ab}, s_{a\$}\} \\
\text{Startzustand} & & s_0 \\
F & = & \{s_f\} \\
\Sigma & = & T \cup \{\$\} \\
K & = & N \cup T
\end{array}
$$

Beachten Sie, dass wir in S_k nicht alle Kombinationen von Zeichen wählen sondern nur die, die in einer akzeptierenden Konfigurationsfolge erreichbar sind.

Zuerst analysieren wir die Grammatik. Wir können bei dem Nicht-Terminalzeichen S unter Kenntnis der ersten zwei Zeichen sicher sagen, wie es weitergeht. Bei dem Nicht-Terminalzeichen B entscheidet schon das erste Zeichen.

$$
\begin{array}{rcl}
\mathsf{select}(aa,S) & = & aSB \\
\mathsf{select}(ab,S) & = & aB \\
\mathsf{select}(a\$,S) & = & aB \\
\mathsf{select}(b,B) & = & b \\
\mathsf{select}(\$,B) & = & \varepsilon
\end{array}
$$

Wir definieren die Überführungsfunktion.

- Initialisieren: $\delta(s_0, \varepsilon, k_0) = (s_\varepsilon, Sk_0)$
- Mehr Eingabezeichen lesen (lookahead), wenn notwendig.

$$\begin{array}{rcl}
\delta(s_\varepsilon, a, S) & = & (s_a, S) \\
\delta(s_\varepsilon, b, B) & = & (s_b, B) \\
\delta(s_\varepsilon, \$, B) & = & (s_\$, B) \\
\delta(s_a, a, S) & = & (s_{aa}, S) \\
\delta(s_a, b, S) & = & (s_{ab}, S) \\
\delta(s_a, \$, S) & = & (s_{a\$}, S)
\end{array}$$

Beachten Sie, dass wir δ nur für sinnvolle Kombinationen von Zuständen und Nicht-Terminalzeichen definieren. Wenn zum Beispiel im Zustand s_ε ein \$ in der Eingabe und ein S auf Keller ist, dann kann das nicht zu einem akzeptierten Wort führen.

- Anwenden der Ersetzungsregeln auf dem Keller, sobald klar ist welche Ersetzungsregel angewendet wird.

$$\begin{array}{rcl}
\delta(s_{aa}, \varepsilon, S) & = & (s_{aa}, aSB) \\
\delta(s_{ab}, \varepsilon, S) & = & (s_{ab}, aB) \\
\delta(s_{a\$}, \varepsilon, S) & = & (s_{a\$}, aB) \\
\delta(s_b, \varepsilon, B) & = & (s_b, b) \\
\delta(s_\$, \varepsilon, B) & = & (s_\$, \varepsilon)
\end{array}$$

- Terminalzeichen auf Keller entfernen.

$$\begin{array}{rcll}
\delta(s_\varepsilon, a, a) & = & (s_\varepsilon, \varepsilon) & \text{Eingabe gegen Keller} \\
\delta(s_\varepsilon, b, b) & = & (s_\varepsilon, \varepsilon) & \\
\delta(s_{aa}, \varepsilon, a) & = & (s_a, \varepsilon) & \text{Zustand gegen Keller} \\
\delta(s_{ab}, \varepsilon, a) & = & (s_b, \varepsilon) & \\
\delta(s_{a\$}, \varepsilon, a) & = & (s_\$, \varepsilon) & \\
\delta(s_b, \varepsilon, b) & = & (s_\varepsilon, \varepsilon) &
\end{array}$$

- Akzeptieren: $\delta(s_\$, \varepsilon, k_0) = (s_f, k_0)$

Wir versuchen mit dem Kellerautomaten das Wort *aaabb*\$ zu akzeptieren.

$$\begin{array}{llllll}
(s_0, aaabb\$, k_0) & \vdash & (s_\varepsilon, aaabb\$, Sk_0) & \vdash & (s_a, aabb\$, Sk_0) & \vdash \\
(s_{aa}, abb\$, Sk_0) & \vdash & (s_{aa}, abb\$, aSBk_0) & \vdash & (s_a, abb\$, SBk_0) & \vdash \\
(s_{aa}, bb\$, SBk_0) & \vdash & (s_{aa}, bb\$, aSBBk_0) & \vdash & (s_a, bb\$, SBBk_0) & \vdash \\
(s_{ab}, b\$, SBBk_0) & \vdash & (s_{ab}, b\$, aBBBk_0) & \vdash & (s_b, b\$, BBBk_0) & \vdash \\
(s_b, b\$, bBBk_0) & \vdash & (s_\varepsilon, b\$, BBk_0) & \vdash & (s_b, \$, BBk_0) & \vdash \\
(s_b, \$, bBk_0) & \vdash & (s_\varepsilon, \$, Bk_0) & \vdash & (s_\$, \varepsilon, Bk_0) & \vdash \\
(s_\$, \varepsilon, k_0) & \vdash & (s_f, \varepsilon, k_0) & & &
\end{array}$$

Das Wort *aaabb*\$ wird akzeptiert.

Kapitel 7

Allgemeinere Chomsky-Sprachen

7.1 Sprachen vom Chomsky-Typ 1

Zur Erinnerung: Eine Grammatik heißt *kontextsensitiv*, wenn alle Regeln die Form

$$\alpha A \beta \to \alpha \gamma \beta \qquad \text{mit } \gamma \neq \varepsilon$$

haben. Sie heißt *monoton*, wenn alle Regeln die Form

$$\alpha \to \beta \qquad \text{mit } |\alpha| \leq |\beta|, \alpha, \beta \in (N \cup T)^*$$

haben. Sie heißt vom *Typ 1*, wenn sie monoton oder kontextsensitiv ist.

Die Menge der kontextsensitiven Sprachen enthält die Menge der kontextfreien Sprachen, denn jede kontextfrei Regel kann auch kontextsensitiv mit *beliebigem* α und β formuliert werden.

In beiden Definitionen darf die Regel $S \to \varepsilon$ vorhanden sein, wenn S nicht auf der rechten Seite einer anderen Regel auftritt. Letzteres lässt sich durch Einführung eines zusätzlichen Symbols immer erreichen (siehe Seite 48).

Satz 7.1. Jede kontextsensitive Grammatik ist auch monoton und zu jeder monotonen gibt es eine äquivalente, kontextsensitive Grammatik.

Beispiel 7.1. Monotone Grammatik für $L = \{a^n b^n c^n \mid n = 1, 2, \ldots\}$.

$$\begin{aligned} S &\to aAbc|abc \\ A &\to aAbC|abC \\ Cb &\to bC \\ Cc &\to cc \end{aligned}$$

Wir betrachten die Ableitung von $a^3b^3c^3$:

$$\begin{array}{llllll} S & \Rightarrow & a\underline{A}bc & \Rightarrow & aa\underline{A}bCbc & \\ & \Rightarrow & a^3bCb\underline{Cb}c & \Rightarrow & a^3bCbbCc & \dots \\ & \Rightarrow & a^3b^3C\underline{Cc} & \Rightarrow & \dots & \\ & \Rightarrow & a^3b^3c^3 & & & \end{array}$$

Für eine äquivalente kontextsensitive Form muss die Regel $Cb \to bC$ geändert werden und die anderen Regeln entsprechend angepasst werden.

$$\begin{array}{rcl} S & \to & aABc|abc \\ A & \to & aABC|abC \\ CB & \to & HB \quad HB \to HC \quad HC \to BC \\ Cc & \to & cc \\ B & \to & b \end{array}$$

Es ergibt sich folgende Ableitung von $a^3b^3c^3$.

$$\begin{array}{llllll} S & \Rightarrow & a\underline{A}Bc & \Rightarrow & aa\underline{A}BCBc & \\ & \Rightarrow & a^3bCB\underline{CB}c & \Rightarrow & a^3bCBHBc & \\ & \Rightarrow & a^3bCBHCc & \Rightarrow & a^3bCBBCc & \dots \\ & \Rightarrow & a^3bBBC\underline{Cc} & \dots & & \\ & \Rightarrow & a^3b^3c^3 & & & \end{array}$$

Die Sprache $L = \{a^nb^nc^n \mid n = 1,2,\dots\}$ ist also vom Typ 1.

Da die Sprache $L = \{a^nb^nc^n \mid n = 1,2,\dots\}$ vom Typ 1 und wir aus Satz 4.9 wissen, dass L nicht vom Typ 2 ist, haben wir ein Beispiel einer kontextsensitiven aber nicht kontextfreien Sprache.

Satz 7.2. Die Familie L_1 der kontextsensitiven Sprachen ist eine *echte* Obermenge der kontextfreien Sprachen L_2.

7.2 Sprachen vom Chomsky-Typ 0

Alle Regeln haben die Form

$$\alpha \to \beta \qquad \text{mit } \alpha, \beta \in (N \cup T)^*$$

ohne sonstige Einschränkungen. Offensichtlich sind alle Sprachen vom Typ > 0 auch vom Typ 0. Dass auch Sprachen existieren, die vom Typ 0 sind aber nicht vom Typ > 0, kann mit einen rein mathematischen Existenzbeweis (Diagonalschluss) gezeigt werden.

Die intuitive Vorgehensweise, eine Sprache vom Typ 0 dadurch zu finden, dass man eine Grammatik vom Typ 0, die nicht vom Typ 1 ist, aufschreibt führt meist nicht zu einem befriedigendem Ergebnis. Es stellt sich bei der Untersuchung der Spracheigenschaften in der Regel heraus, dass die zugehörige Sprache auch von einer Grammatik vom Typ 1 oder höher erzeugt werden kann.

Satz 7.3. (Folgerung aus Existenzbeweis). Die Familie L_0 der allgemeinen Sprachen ist eine echte Obermenge der kontextsensitiven beziehungsweise monotonen Sprachen L_1.

Satz 7.4. Die Familie L_0 der allgemeinen Sprachen ist abzählbar.

Man beachte, dass die Menge *aller Sprachen* über einem Alphabet nicht abzählbar ist.

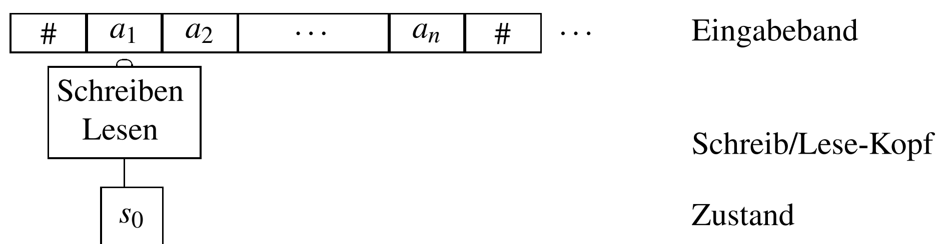
Kapitel 8

Turing-Maschinen und allgemeinere Chomsky-Sprachen

8.1 Turing-Maschinen

Turing beschrieb die nach ihm benannten Maschinenmodelle 1936 im Zusammenhang mit dem Problem der Berechenbarkeit von Funktionen und der Frage, was man unter einem Algorithmus zu verstehen hat. Die Turing-Maschine gibt einen algorithmischen Zugang zu dem von Gödel 1931 veröffentlichten Unvollständigkeitssatz, der zeigt, dass es Aussagen gibt, die nicht entschieden werden können.

These 8.1. (These von Turing) Jeder Algorithmus lässt sich als Turing-Maschine darstellen. Jede Turing-Maschine ist ein Algorithmus.



Eine Turing-Maschine arbeitet in einfachster Form mit einem einseitig unendlichen Band. Das Band besitzt Felder mit jeweils einem Zeichen, welche über einen Schreib/Lese-Kopf ausgegeben oder eingelesen werden können.

Der typische elementare Arbeitsschritt einer Turing-Maschine besteht aus der Ausführung folgender Aktionen. Erst wird das aktuelle Zeichen gelesen. In Abhängigkeit dieses Zeichens und dem aktuellen Zustand wird das gleiche oder ein anderes Zeichen in das aktuelle Feld geschrieben. Anschließend wird dann der Schreib/Lese-Kopf gegebenenfalls auf ein Nachbarfeld positioniert.

Definition 8.1. $TM = (S, s_0, F, \Sigma, B, \delta)$ ist eine (deterministische) Turing-Maschine, wenn für die einzelnen Komponenten gilt:

S Endliche Menge der Zustände der Turing-Maschine

s_0 Interner Anfangszustand, $s_0 \in S$

F Menge der Endzustände, $F \subseteq S$

Σ Endliche Menge der Eingabezeichen

B Endliche Menge der Bandzeichen (einschließlich Σ und eines Leerzeichens #, das nicht als Eingabezeichen eines Wortes auftreten darf)

δ (Determinierte) Überführungsfunktion $\delta : S \times B \to S \times B \times X$, wobei $X = \{l, r, h\}$ die möglichen Bewegungen des Schreib/Lese-Kopfs darstellt

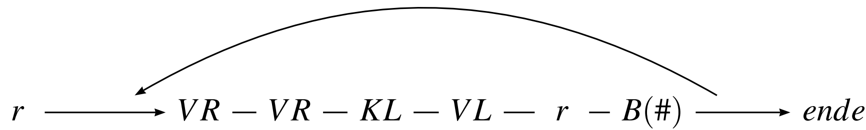
Die Tabelle von δ wird auch als *Programm* der Turing-Maschine bezeichnet.

Beispiel 8.1. Wir geben im Folgenden einige Beispiele für elementare Turing-Maschinen. Für immer wiederkehrende Aufgaben wie „rechtes Wortende finden“, „linkes Wortende finden“, „Zeichen kopieren“, ... ist es sinnvoll, elementare Turing-Maschinen (oder Turing-Programme) zur Verfügung zu haben.

(r)	Ein Zeichen nach rechts	$\delta(s_0, x) = (s_f, x, r)$ für beliebiges Bandzeichen x
(L)	Linkes Wortende suchen	$\delta(s_0, x) = (s_0, x, l)$ für $x \neq \#$, $\delta(s_0, \#) = (s_f, \#, h)$
(VR)	Aktuelles Zeichen a auf das erste Freizeichen rechts verschieben (Für jedes Eingabezeichen a wird dazu ein innerer Zustand s_a benötigt)	$\delta(s_0, a) = (s_a, \#, r)$, $\delta(s_a, x) = (s_a, x, r)$ für $x \neq \#$, $\delta(s_a, \#) = (s_f, a, h)$
(KL)	Aktuelles Zeichen a auf das erste Freizeichen links kopieren	$\delta(s_0, a) = (s_a, a, l)$, $\delta(s_a, x) = (s_a, x, l)$ für $x \neq \#$, $\delta(s_a, \#) = (s_f, a, h)$
(B(#))	Bedingtes Anhalten in Abhängigkeit vom Feldinhalt #	$\delta(s_0, x) = (s_{f_1}, x, h)$ für $x \neq \#$, $\delta(s_0, \#) = (s_{f_2}, \#, h)$

Solche einfachen Programme können dann für komplexere Aufgaben zusammengesetzt werden. Dabei ist der Anfangszustand s_0 des nächsten Programms gleich dem Endzustand s_f des vorhergehenden Programms zu setzen.

Beispiel 8.2. Für das *„Kopieren eines Wortes“* ergibt sich die folgende zusammengesetzte Turing-Maschine.



Der Schreib/Lese-Kopf steht dabei im Ausgangszustand s_0 auf dem ersten Freizeichen (#) links vom Eingabewort. Es ergibt sich folgendes *Programm* für das Kopieren eines Wortes aus $\{a,b,c\}^*$.

δ	a	b	c	#	Unterprogramm
s_0	–	–	–	$(s_1,\#,r)$	r
s_1	$(s_{1_a},\#,r)$	$(s_{1_b},\#,r)$	$(s_{1_c},\#,r)$	$(s_9,\#,h)$	VR, B(#)
s_{1_a}	(s_{1_a},a,r)	(s_{1_a},b,r)	(s_{1_a},c,r)	(s_2,a,h)	
s_{1_b}	(s_{1_b},a,r)	(s_{1_b},b,r)	(s_{1_b},c,r)	(s_2,b,h)	
s_{1_c}	(s_{1_c},a,r)	(s_{1_c},b,r)	(s_{1_c},c,r)	(s_2,c,h)	
s_2	$(s_{2_a},\#,r)$	$(s_{2_b},\#,r)$	$(s_{2_c},\#,r)$	–	VR
s_{2_a}	(s_{2_a},a,r)	(s_{2_a},b,r)	(s_{2_a},c,r)	(s_3,a,h)	
s_{2_b}	(s_{2_b},a,r)	(s_{2_b},b,r)	(s_{2_b},c,r)	(s_3,b,h)	
s_{2_c}	(s_{2_c},a,r)	(s_{2_c},b,r)	(s_{2_c},c,r)	(s_3,c,h)	
s_3	(s_{3_a},a,l)	(s_{3_b},b,l)	(s_{3_c},c,l)	–	KL
s_{3_a}	(s_{3_a},a,l)	(s_{3_a},b,l)	(s_{3_a},c,l)	(s_4,a,h)	
s_{3_b}	(s_{3_b},a,l)	(s_{3_b},b,l)	(s_{3_b},c,l)	(s_4,b,h)	
s_{3_c}	(s_{3_c},a,l)	(s_{3_c},b,l)	(s_{3_c},c,l)	(s_4,c,h)	
s_4	$(s_{4_a},\#,l)$	$(s_{4_b},\#,l)$	$(s_{4_c},\#,l)$	–	VL
s_{4_a}	(s_{4_a},a,l)	(s_{4_a},b,l)	(s_{4_a},c,l)	(s_5,a,h)	
s_{4_b}	(s_{4_b},a,l)	(s_{4_b},b,l)	(s_{4_b},c,l)	(s_5,b,h)	
s_{4_c}	(s_{4_c},a,l)	(s_{4_c},b,l)	(s_{4_c},c,l)	(s_5,c,h)	
s_5	(s_1,a,r)	(s_1,b,r)	(s_1,c,r)	$(s_9,\#,h)$	r, B(#)

In obigen Beispiel haben wir eine *Programmiertechnik* für Turing-Maschinen eingesetzt. Wir haben die Zustände als Zwischenspeicher für Informationen verwendet. Diese Technik wird von sehr vielen Turing-Programmen verwendet.

Definition 8.2. Unter einer *Konfiguration* einer Turing-Maschine versteht man eine Zeichenkette $\alpha s \beta \in B^* \times S \times B^*$. Dabei bedeuten

- $\alpha, \beta \in B^*$ die beiden Teile des Bandinhalts, die sich ergeben, wenn der Schreib/Lese-Kopf auf dem ersten Zeichen von β steht.
- $s \in S$ der interne Zustand, in dem sich die Turing-Maschine gerade befindet.

Beispiel 8.3. Wir spielen die Folge der Konfigurationen beim Kopieren des Wortes *abbc* mit der in Beispiel 8.2 beschriebenen Turing-Maschine durch. $s_0\#abbc\#\#$ ist die Startkonfiguration. Die Turing-Maschine befindet sich im Zustand s_0 und der Schreib/Lese-Kopf zeigt auf das #-Zeichen am Anfang. Es ergibt sich die folgende Konfigurationsfolge für die ersten beiden Durchläufe.

Erster Durchlauf

s_0	#	a	b	b	c	#	#	#	#	#	
*	s_1	a	b	b	c	#	#	#	#	#	(r)
*	#	s_{1_a}	b	b	c	#	#	#	#	#	(VR)
*	#	b	s_{1_a}	b	c	#	#	#	#	#	
*	#	b	b	s_{1_a}	c	#	#	#	#	#	
*	#	b	b	c	s_{1_a}	#	#	#	#	#	
*	#	b	b	c	s_2	a	#	#	#	#	
*	#	b	b	c	#	s_{2_a}	#	#	#	#	(VR)
*	#	b	b	c	#	s_3	a	#	#	#	
*	#	b	b	c	s_{3_a}	#	a	#	#	#	(KL)
*	#	b	b	c	s_4	a	a	#	#	#	
*	#	b	b	s_{4_a}	c	#	a	#	#	#	(VL)
*	#	b	s_{4_a}	b	c	#	a	#	#	#	
*	#	s_{4_a}	b	b	c	#	a	#	#	#	
*	s_{4_a}	#	b	b	c	#	a	#	#	#	
*	s_5	a	b	b	c	#	a	#	#	#	

Zweiter Durchlauf

#	a	s_1	b	b	c	#	a	#	#	#	(r)
#	a	#	s_{1_b}	b	c	#	a	#	#	#	(VR)
#	a	#	b	s_{1_b}	c	#	a	#	#	#	
#	a	#	b	c	s_{1_b}	#	a	#	#	#	
#	a	#	b	c	s_2	b	a	#	#	#	
#	a	#	b	c	#	s_{2_b}	a	#	#	#	(VR)
#	a	#	b	c	#	a	s_{2_b}	#	#	#	
#	a	#	b	c	#	a	s_3	b	#	#	
#	a	#	b	c	#	s_{3_b}	a	b	#	#	(KL)
#	a	#	b	c	s_{3_b}	#	a	b	#	#	
#	a	#	b	c	s_4	b	a	b	#	#	
#	a	#	b	s_{4_b}	c	#	a	b	#	#	(VL)
#	a	#	s_{4_b}	b	c	#	a	b	#	#	
#	a	s_{4_b}	#	b	c	#	a	b	#	#	
#	a	s_5	b	b	c	#	a	b	#	#	

Wenn wir von einer Konfiguration $\alpha s \beta$ in einem Schritt in eine Konfiguration $\alpha' s' \beta'$ kommen, dann schreiben wir auch

$$\alpha s \beta \vdash \alpha' s' \beta' \quad .$$

Wenn wir von einer Konfiguration $\alpha s \beta$ in keinem, einem oder mehreren Schritten in eine Konfiguration $\alpha' s' \beta'$ kommen, dann schreiben wir auch

$$\alpha s \beta \vdash^* \alpha' s' \beta' \quad .$$

8.2 Turing-Maschinen als Akzeptoren

In der Ausgangssituation steht ein Wort w aus Σ^*, eingegrenzt durch zwei Leerzeichen (#), auf dem Band. Die Maschine ist im Anfangszustand s_0 und der Schreib/Lese-Kopf steht auf dem Leerzeichen am Anfang des Bandes.

Definition 8.3. Das Wort w wird von der Turing-Maschine *TM akzeptiert*, wenn nach einer Folge von Konfigurationen eine Endkonfiguration entsteht, bei der sich *TM* in einem internen Endzustand $s_f \in F$ befindet.

$$s_0\#w\#\cdots \vdash^* \alpha s_f \beta$$

Auf den in der Endkonfiguration vorhandenen Bandinhalt kommt es nicht an. Unter der *Sprache einer Turing-Maschine* $L(TM)$ versteht man alle Worte, die von *TM* akzeptiert werden.

Beispiel 8.4. Eine Turing-Maschine für die Sprache $L = \{a^n b^n c^n \mid n = 1, 2, \ldots\}$.

$\Sigma = \{a, b, c\}$

$B = \{a, b, c, \#, A, B, C\}$

$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$

$F = \{s_6\}$

$\delta =$

	a	b	c	A	B	C	$\#$
s_0	–	–	–	–	–	–	$(s_1, \#, r)$
s_1	(s_2, A, r)	–	–	–	(s_5, B, r)	–	–
s_2	(s_2, a, r)	(s_3, B, r)	–	–	(s_2, B, r)	–	–
s_3	–	(s_3, b, r)	(s_4, C, l)	–	–	(s_3, C, r)	–
s_4	(s_4, a, l)	(s_4, b, l)	–	(s_1, A, r)	(s_4, B, l)	(s_4, C, l)	–
s_5	–	–	–	–	(s_5, B, r)	(s_5, C, r)	$(s_6, \#, h)$

Eine Turing-Maschine kann alle Sprachen akzeptieren, die von einer Grammatik vom Typ 0 erzeugt werden.

Satz 8.1. Zu jeder Sprache $L(G)$ einer Grammatik vom Typ 0 gibt es eine Turing-Maschine *TM* mit $L(G) = L(TM)$ und umgekehrt.

Satz 8.2 (Wortproblem bei Turing-Maschinen)**.** Es ist nicht entscheidbar, ob eine beliebige als Akzeptor entworfenen Turing-Maschine bei der Abarbeitung eines beliebigen Wortes anhält oder nicht.

8.3 Erweiterungen der Turing-Maschine

Im Folgenden stellen wir einige Erweiterungen des Modells einer Turing-Maschine vor. Diese Erweiterungen erhöhen die Mächtigkeit des Modells allerdings nicht.

Definition 8.4. $TM = (S, s_0, F, \Sigma, B, \delta)$ ist eine *nicht-deterministische* Turing-Maschine, wenn die einzelnen Komponenten bis auf δ die gleiche Bedeutung haben wie bei der deterministischen Turing-Maschine. Die Überführungsfunktion δ bildet in eine Menge von Zielen ab. Das heißt also für die Überführungsfunktion gilt

$$\delta : S \times B \to P(S \times B \times X) \ .$$

Satz 8.3. Zu jeder als Akzeptor entworfenen nicht-deterministischen Turing-Maschine gibt es eine deterministische, die die gleiche Sprache akzeptiert.

Zum Verständnis der Äquivalenz machen Sie sich klar, dass eine deterministische Turing-Maschine sich alle möglichen Konfigurationen einer nicht-deterministischen Turing-Maschine auf dem Band speichern kann. Der sich aufbauende Baum von Möglichkeiten wird dann mit Hilfe einer Warteschlange abgearbeitet. Damit werden alle möglichen Konfigurationen durchlaufen. Die deterministische Turing-Maschine simuliert alle möglichen Konfigurationsfolgen einer nicht-deterministischen Turing-Maschine.

Mehr eine Programmiertechnik als eine Erweiterung ist die Verwendung von Spuren. Wir stellen uns das Eingabeband in mehrere Spuren unterteilt vor. Statt dem Eingabe-Alphabet B haben wir nun ein Eingabe-Alphabet B^k. Jedes Tupel aus B^k ist dann ein Zeichen aus dem Eingabe-Alphabet. Die Turing-Maschine hat dann immer noch ein Band.

Die Verwendung von Spuren erlaubt es uns auch die Beschränkung aufzuheben, dass das Band nur einseitig unendlich ist. Wir können ein beidseitig unendliches Band zulassen und dieses durch zwei Spuren simulieren. Mit einem Extra-Zeichen merken wir uns den Punkt am Anfang des Bandes an dem wir umdrehen müssen.

Definition 8.5. Bei einer *Turing-Maschine mit mehreren Bändern* wird statt mit nur einem Band mit mehreren gleichartigen Bändern gearbeitet. Die Überführungsfunktion hat dabei die Form $\delta : S \times B^k \to S \times B^k \times X^k$.

Turing-Maschinen mit mehreren Bändern arbeiten effizienter als Turing-Maschinen mit nur einem Band.

Satz 8.4. Zu jeder als Akzeptor entworfenen Turing-Maschine mit mehreren Bändern gibt es eine mit nur einem Band, die die gleiche Sprache akzeptiert.

Zum Verständnis der Äquivalenz machen Sie sich klar, dass man eine Turing-Maschine mit mehreren Bändern mit einer Turing-Maschine mit einem Band simulieren kann. Dazu unterteilen wir das eine Band in $2k$ Spuren, wenn wir k Bänder simulieren wollen. Wir haben also für jedes Band zwei Spuren zu Verfügung. In der einen Spur speichern wir den Inhalt. In der anderen Spur speichern wir die aktuelle Bandposition mit einem speziellem Zeichen. In den Zuständen speichern wir uns je simuliertem Band in welche Richtung wir laufen müssen, um die aktuelle Position des Bands zu erreichen.

Eine deterministische Turing-Maschine mit einem einseitig unendlichen Band mit einer Spur ist also gleich mächtig wie eine nicht-deterministische Turing-Maschine mit mehreren Bändern.

8.4 Linear beschränkte Automaten

Bei linear beschränkten Automaten (LBA) handelt es sich um als Akzeptoren entworfene Turing-Maschinen. Allerdings unterliegen diese der Beschränkung, dass die bei der Analyse zur Verfügung stehende Länge des Bandes linear abhängig ist von der Länge des zu untersuchenden Eingabewortes beziehungsweise dieser Länge entspricht.

Satz 8.5. Zu jeder Sprache $L(G)$ einer Grammatik vom Typ 1 (monotone Grammatiken) gibt es einen nicht-deterministischen LBA mit $L(G) = L(LBA)$ und umgekehrt.

Ob *deterministische* linear beschränkte Automaten die gleiche Mächtigkeit als Akzeptoren haben wie die *nicht-deterministischen*, ist eine Frage, die im Gegensatz zu den anderen behandelten Automatentypen bisher noch nicht beantwortet werden konnte.

Das *Wortproblem* für LBAs ist entscheidbar, das heißt für jedes Wort und jeden LBA ist entscheidbar, ob der LBA das Wort akzeptiert oder nicht.

Kapitel 9

Entscheidbarkeit und Berechenbarkeit

In diesem Kapitel geht es um die Frage, welche Probleme von Rechnern überhaupt gelöst werden können. Da zur Problemlösung eine Eingabe in eine Ausgabe überführt wird, geht es im allgemeinen um die Berechenbarkeit von Funktionen verschiedenen Typs, wie zum Beispiel folgende Funktionen:

$f : \Sigma^* \to \Sigma^*$	Wortfunktionen über endlichem Alphabet Σ
$f : \Sigma^* \to \mathbb{N}$	$\mathbb{N} =$ Menge der natürlichen Zahlen $0, 1, 2, \ldots$
$f : \mathbb{N} \to \mathbb{N}$	Funktionen innerhalb der Menge natürlicher Zahlen
$f : \mathbb{N}^n \to \mathbb{N}$	Funktionen mit mehreren Argumenten

Definition 9.1. Eine Funktion heißt *TM-berechenbar*, wenn es eine Turing-Maschine gibt, die mit dem Argument als Inhalt des Eingabebandes startet, die Eingabe verarbeitet und mit dem Funktionswert auf dem Eingabeband endet.

Es wird allgemein angenommen, dass TM-Berechenbarkeit ausreichend ist, um alle möglichen Berechnungen durchzuführen. Zumindest all das, was heute ein Rechner tun kann, ist abgedeckt.

These 9.1 (Church-Turing-These)**.**
Jede intuitiv berechenbare Funktion ist auch TM-berechenbar.

Definition 9.2. Eine Sprache L (Menge von Worten) über einem Alphabet Σ heißt *entscheidbar*, wenn es einen Algorithmus gibt, der für jedes Wort $w \in \Sigma^*$ mit der Feststellung, ob das Wort zur Sprache gehört oder nicht, stoppt.

Definition 9.3. (äquivalente Definition zu Definition 9.2) L heißt entscheidbar, wenn die *charakteristische Funktion*

$$cf(x) = \begin{cases} 1 & \text{für} \quad x \in L \\ 0 & \text{für} \quad x \notin L \end{cases}$$

berechenbar ist.

Satz 9.1. Die Sprachen vom Chomsky-Typ 1 sind entscheidbar.

Beweis. Wir beweisen den Satz durch Angabe eines Algorithmus, der die Wortlänge des zu testenden Wortes nutzt. Sei x das zu untersuchende Wort mit $|x| = n$, $Abl(R)$ alle Ableitungen der Satzformen in R und S das Startsymbol, dann ist wortinL ein Entscheidungsalgorithmus.

```
1  def wortinL(x):
2      R,T = ∅,{S}
3      while x ∉ T and R ≠ T: # nicht gefunden oder Fortschritt
4          R = T
5          T = R ∪ {w ∈ Abl(R) | |w| ≤ n}
6      return x ∈ T
```

Eine andere Beweismöglichkeit ist die Konstruktion einer Turing-Maschine, die nach dem Bottom-Up-Prinzip und mit Hilfe der Grammatik vom Typ 1 ein Wort der Sprache, das auf dem Eingabeband steht nicht-deterministisch auf das Startsymbol reduziert. □

Definition 9.4. Eine Sprache L über einem Alphabet Σ heißt *semi-entscheidbar*, wenn es einen Algorithmus gibt, der für genau die Worte $w \in L$ mit dem Ergebnis stoppt, dass sie zur Sprache gehören. Für Worte $w \notin L$ kann der Algorithmus entweder mit der Feststellung stoppen, dass das Wort nicht zur Sprache gehört, oder auch nicht stoppen.

Definition 9.5. (äquivalent zu Definition 9.4) L heißt semi-entscheidbar, wenn die charakteristische Funktion $cf(x) = 1$ für $x \in L$ auf L berechenbar ist.

Satz 9.2. Eine Sprache $L \subseteq \Sigma^*$ ist genau dann entscheidbar, wenn sowohl L als auch ihr Komplement $\overline{L}$ semi-entscheidbar sind.

Beweis. Wenn L entscheidbar dann ist klar, dass L und $\overline{L}$ semi-entscheidbar sind. Es bleibt zu zeigen, dass wenn L und $\overline{L}$ semi-entscheidbar sind, dann ist L entscheidbar. Sei semiL der Algorithmus für die Semi-Entscheidbarkeit von L und semiKompL der Algorithmus für die Semi-Entscheidbarkeit von $\overline{L}$. Wir wissen nur, dass wenn ein Wort in der Sprache ist, dann hält der Algorithmus, sonst könnte er unendlich lange laufen.

Ein Algorithmus kann schrittweise ausgeführt werden. Wir können also die Algorithmen semiL und semiKompL so abwandeln, dass sie maximal **max** Schritte (zweiter Parameter) laufen. Wenn die Entscheidung noch nicht getroffen ist geben wir vielleicht zurück, ansonsten die Entscheidung True oder False . Dies führt zu einem Entscheidungsverfahren.

```
1  def wortinL(w):
2      i = 1
3      while True:
4          if semiL(w, i) ≠ vielleicht :
```

```
5             return semiL(w, i)
6         if semiKompL(w, i) ≠ vielleicht:
7             return semiKompL(w, i)
8         i += 1
```

Beachten Sie, dass die **while**-Schleife nicht unendlich lange laufen kann, da jedes Wort entweder in L oder nicht in L ist, also entweder semiL oder semiKompL in endlichen vielen Schritten i (beliebig) anhalten muss. □

Satz 9.3. Das Wortproblem von Sprachen des Typs 0 ist semi-entscheidbar.

Beweis. Wir haben in Kapitel 8 gesehen, dass es zu jeder Sprache L vom Typ 0 eine Turing-Maschine TM gibt, die L akzeptiert. □

Die Menge der semi-entscheidbaren Sprachen entspricht damit der Menge der von Turing-Maschinen akzeptierten Sprachen.

9.1 Nicht entscheidbare Probleme

Wir zeigen informell, dass es gewisse Probleme gibt für die es keinen Algorithmus gibt. In Abschnitt 9.2 werden wir dies formal mit Hilfe von speziellen Turing-Maschinen zeigen.

Nehmen wir an wir haben ein Entscheidungsproblem. Wenn wir eine Funktion realisieren, die dieses Problem löst, dann erwarten wir von der Funktion eine Antwort, zum Beispiel einen booleschen Wert True oder False . Bei dem in der Funktionsdefinition realisierten Algorithmus kann es eventuell passieren, dass der gewählte Algorithmus unendlich lange läuft. Zu jedem beliebigen Zeitpunkt können wir uns nicht immer sicher sein, ob das Programm noch zu einem Ergebnis kommt oder nicht.

Nehmen wir als Beispiel „Fermats letzte Behauptung[1]“.

$$\forall n \in \mathbb{N}, n > 2 : \nexists x, y, z \in \mathbb{N} : x^n + y^n = z^n$$

Ein Algorithmus, der eine Lösung sucht, ist schnell geschrieben.

```
1 grenze = 1
2 while True:
3     for n ∈ {3,...,grenze}:
4         for x ∈ {1,...,grenze}:
5             for y ∈ {1,...,grenze}:
6                 for z ∈ {1,...,grenze}:
7                     if x^n + y^n == z^n:
8                         return True
9     grenze += 1
```

[1] inzwischen nach allgemeiner Ansicht wohl doch „Fermats letzter Satz“

Die Frage ist, ob dieser Algorithmus anhält mit dem Ergebnis True oder nicht anhält. Wenn der Algorithmus nicht anhält, dann ist „Fermats letzte Behauptung“ wahr, ansonsten falsch. Es hat die Menschheit rund 300 Jahre gekostet, um für ein recht triviales Programm zu entscheiden, ob es anhält oder nicht.

Im allgemeinen kann man nicht entscheiden, ob ein beliebiges Programm mit einer gegebenen Eingabe anhält oder nicht. Wir können uns dies mit einem Widerspruchsargument klarmachen.

Nehmen wir an, wir könnten entscheiden, ob ein beliebiges Programm P mit einer gegebenen Eingabe E anhält. Dann gibt es einen Entscheidungsalgorithmus dafür. Dieser Algorithmus hält immer in endlich vielen Schritten an, sonst wäre es ja kein Entscheidungsalgorithmus. Diesen Algorithmus können wir in einer beliebigen „Turing-mächtigen“ Programmiersprache implementieren (C, Java, Python, ...). Nehmen wir an, wir haben dies getan und haben jetzt eine entsprechende Funktion haelt in unserer Lieblingsprogrammiersprache definiert. Es gilt Folgendes.

$$\mathsf{haelt}(P,E) = \begin{cases} \mathsf{True} & \text{falls } P \text{ mit } E \text{ hält} \\ \mathsf{False} & \text{sonst} \end{cases}$$

Wir schreiben nun ein neues Programm weird das haelt beinhaltet.

```
1 def weird(P):
2     if haelt(P, P):
3         while True: # Endlosschleife
4             pass
5     return True # wenn haelt(P, P) nicht gilt
```

Wir nehmen dazu ohne Beschränkung der Allgemeinheit an, dass sowohl P als auch die Eingabe irgendwie – zum Beispiel als Bitfolgen – kodiert sind. Es ist also kein Problem als Eingabe E ein Programm P zu verwenden. Falls also das Programm P mit sich selbst als Eingabe anhält, dann läuft weird unendlich lange und hält also nicht. Falls das Programm P mit sich selbst als Eingabe nicht anhält, dann stoppt weird mit dem Ergebnis True. Bei gegebenem Programm haelt lässt sich unser neues Programm weird einfach implementieren und nutzt normale Möglichkeiten von Programmiersprachen.

Sei P_{weird} die Kodierung des vollständigen (inklusive haelt) Programms weird. Wird der Aufruf

$$\mathsf{weird}(P_{\mathsf{weird}})$$

ein Ergebnis zurückgeben, und damit halten, oder unendlich lange laufen? Wir spielen beide Ablaufmöglichkeiten durch:

- Nehmen wir zuerst an, dass das Programm weird mit sich selbst als Eingabe hält. Nach Definition von haelt ist dann das Ergebnis des Aufrufs $\mathsf{haelt}(P_{\mathsf{weird}},P_{\mathsf{weird}})$ True. Aber dann läuft weird laut der Programmierung von weird unendlich lange und hält nicht; ein Widerspruch.

- Nehmen wir also an, dass das Programm weird mit sich selbst als Eingabe nicht hält. Nach Definition von haelt ist das Ergebnis des Aufrufs haelt(P_{weird},P_{weird}) False. Aber dann stoppt weird laut der Programmierung von weird mit dem Ergebnis True; ein Widerspruch.

Wir haben eine paradoxe Situation. Also muss unsere Annahme falsch sein. Unsere einzige Annahme war, dass es ein Entscheidungsverfahren gibt, das entscheidet, ob ein beliebiges Programm mit einer beliebigen Eingabe hält. Aufgrund des Widerspruchs folgern wir:

Es gibt kein Programm (Entscheidungsverfahren), das für ein beliebiges Programm und eine gegebene Eingabe ermittelt, ob das Programm mit dieser Eingabe anhält oder nicht.

Die Fragestellung ob ein Programm mit gegebener Eingabe anhält wird auch das *Halteproblem* genannt. Das Halteproblem ist *nicht entscheidbar*.

9.2 Nicht entscheidbare Sprachen

Wir weisen jetzt die Existenz nicht entscheidbarer Sprachen nach. Betrachten Sie dazu die Menge aller Turing-Maschinen mit

$\Sigma = \{0,1\}, B = \{0,1,\#\}$

$Z = \{z_1, z_2, \ldots, z_n\}$, z_1 sei Anfangszustand, $F = \{z_n\}$

Wir wollen jeder solchen Turing-Maschine eine eindeutig bestimmte Bitfolge als Codewort zuordnen. Dabei kann man zum Beispiel die folgende Codierung benutzen:

Zeichen	Zustände	Richtung
$0 \to 00$	$z_i \to \overbrace{00\ldots0}^{i-\text{mal } 0}$	$r \to 01$
$1 \to 01$		$l \to 10$
$\# \to 10$		$h \to 00$

Mit dieser Codierung wird der Wert $\delta(z_2,0) = (z_3,\#,r)$ der Überführungsfunktion folgendes codiertes Wort ergeben.

$$\delta(z_2,0) = (z_3,\#,r): \quad \overbrace{00}^{z_2} \ \$ \ \overbrace{00}^{0} \ \$ \ \overbrace{000}^{z_3} \ \$ \ \overbrace{10}^{\#} \ \$ \ \overbrace{01}^{r} \quad (\$ \text{ Trennung})$$

$$00 \ 1 \ 00 \ 1 \ 000 \ 1 \ 10 \ 1 \ 01 \quad (1 \text{ Trennung})$$

Nur bei der Codierung der Zustände wird eine flexible Anzahl von Zeichen verwendet, jedoch nur 0en. Wenn man als Trennzeichen eine 1 verwendet bleibt die Codierung eindeutig und rekonstruierbar. Die Menge aller möglichen Werte der Überführungsfunktion

bilden eine reguläre Sprache gemäß folgendem Ausdruck für jede Definition der Überführungsfunktion $\delta(z_i,b_i) = (z_j,b_j,k_j)$ mit $b_i,b_j \in \{0,1,\#\}$ und $k_j \in \{r,l,h\}$.

$$\overbrace{0(0)^*}^{z_i}1\overbrace{(00|01|10)}^{b_i}1\overbrace{0(0)^*}^{z_j}1\overbrace{(00|01|10)}^{b_j}1\overbrace{(00|01|10)}^{k_j}$$

Als *Codewort* einer Turing-Maschine definieren wir die Konkatenation der codierten Werte der Überführungsfunktion in einer bestimmten Reihenfolge. Die Reihenfolge sowie die Art der Codierung ist für die nachfolgende Herleitung nebensächlich und hätte auch nach anderen Methoden erfolgen können. Es sei nun TML die Menge aller Codeworte der hier betrachteten Turing-Maschinen. TML stellt eine Sprache über $\{0,1\}$ dar. Ist w ein Wort aus TML dann sei TM_w die dazugehörige eindeutig bestimmte Turing-Maschine. Man kann die Frage stellen, ob w von TM_w akzeptiert wird oder grundsätzlich, ob die Sprache

$$STML = \{w \in TML \mid w \in L(TM_w)\} \ ,$$

die Menge aller Turing-Maschinen, die sich selbst akzeptieren, entscheidbar ist. Wir gehen dazu einen Umweg über das Komplement $\overline{STML}$ dieser Sprache das sich zusammensetzt aus den nicht richtig kodierten Worten und allen zwar richtig kodierten Worten aber nicht sich selbst akzeptierenden Turing-Maschinen.

$$\overline{STML} = \{w \in \{0,1\}^* \mid w \notin TML\} \cup \{w \in TML \mid w \notin L(TM_w)\}$$

Satz 9.4. $\overline{STML}$ ist nicht semi-entscheidbar ist.

Beweis. Angenommen $\overline{STML}$ wäre semi-entscheidbar. Dann gibt es eine Turing-Maschine die $\overline{STML}$ akzeptiert. Diese Turing-Maschine hat auch ein Codewort $x \in TML$. Die Sprache dieser Turing-Maschine wäre dann genau $\overline{STML}$, das heißt $L(TM_x) = \overline{STML}$. Für x gibt es nun zwei Möglichkeiten.

1. $x \in \overline{STML}$:

   Nach Definition von $\overline{STML}$ sind in $\overline{STML}$ alle Codierungen von Turing-Maschinen enthalten, die Ihre eigene Kodierung nicht akzeptieren, also die Menge $\{w \in TML \mid w \notin L(TM_w)\}$. Das heißt aber, dass dann $x \notin L(TM_x)$ gelten muss. Da aber $L(TM_x) = \overline{STML}$ wäre dann $x \notin \overline{STML}$. Dies ist ein Widerspruch zu $x \in \overline{STML}$.

2. $x \notin \overline{STML}$:

   Da $\overline{STML}$ das Komplement von $STML$ ist gilt dann $x \in STML$. Daher gilt x akzeptiert sich selbst als Eingabe, also die Menge $\{w \in TML \mid w \in L(TM_w)\}$. Also ist $x \in L(TM_x)$. Da $L(TM_x) = \overline{STML}$ wäre dann $x \in \overline{STML}$. Dies ist ein Widerspruch zu $x \notin \overline{STML}$.

Daraus folgt, dass die Annahme $\overline{STML}$ wäre semi-entscheidbar falsch ist. □

Die Sprache $STML = \{w \in TML \mid w \in L(TM_w)\}$ ist also nicht entscheidbar, weil ihr Komplement $\overline{STML}$ nicht semi-entscheidbar ist.

9.3 Das Halteproblem für Turing-Maschinen

Wir haben uns schon in Abschnitt 9.1 informell mit dem Halteproblem beschäftigt. Wir tun dies nun noch einmal für Turing-Maschinen. Die Frage ist: Gibt es einen Algorithmus, der feststellt, ob eine beliebige Turing-Maschine anhält oder nicht? Wegen der Church-Turing-These ist diese Frage generell für beliebige Algorithmen bedeutsam. Zunächst betrachten wir das *spezielle Halteproblem*.

Gibt es einen Algorithmus, der feststellt, ob eine beliebige Turing-Maschine angesetzt auf ihr eigenes Codewort, anhält oder nicht. Die Frage ist gleichbedeutend mit der Entscheidbarkeit der folgenden Sprache.

$$K = \{w \in TML \mid TM_w \text{ angesetzt auf } w \text{ hält}\}$$

Satz 9.5. Das spezielle Halteproblem (beziehungsweise Selbstanwendbarkeitsproblem) ist nicht entscheidbar.

Beweis. Der Satz ist bewiesen, wenn das Komplement von K

$$\overline{K} = \{w \in \{0,1\}^* \mid w \notin TML\} \cup \{w \in TML \mid TM_w \text{ angesetzt auf } w \text{ hält nicht }\}$$

nicht semi-entscheidbar ist. Dies kann aber in ähnlicher Weise in Satz 9.4 mit einem Widerspruchsbeweis gezeigt werden. □

Das *generelle Halteproblem*, ob eine beliebige TMmit dem Codewort $w \in TML$ bei der Eingabe eines beliebigen Wortes $x \in \{0,1\}^*$ anhält oder nicht, wird durch folgende Sprache repräsentiert.

$$H = \{w\$x \mid TM_w \text{ angesetzt auf } x \text{ hält }\}$$

Satz 9.6. Das generelle Halteproblem ist nicht entscheidbar.

Beweis. Es ist

$$\{w\$w \mid TM_w \text{ angesetzt auf } w \text{ hält }\}$$

eine Teilmenge von H, die mit dem speziellen Halteproblem K identifiziert werden kann.

Wäre H entscheidbar, dann könnte man die charakteristische Funktion von H leicht zu einer charakteristischen Funktion von K umwandeln, weil entscheidbar ist, ob ein Wort $w\$x$ die Form $w\$w$ hat oder nicht. Also wäre auch K entscheidbar. Das ist ein Widerspruch. Also ist H nicht entscheidbar. □

Definition 9.6. Unter der *universellen Turing-Maschine* (UTM) versteht man eine Turing-Maschine TM, die jede Turing-Maschine mit dem Codewort w simulieren kann, wenn das zusammengesetzte Wort $w\$x$ mit $w \in TML$ und $x \in \{0,1\}^*$ auf dem Eingabeband steht.

Definition 9.7. Unter der *universellen Sprache* (U) versteht man die Sprache

$$U = \{w\$x \mid x \in L(TM_w)\} \ .$$

Satz 9.7. Die Sprache U ist nicht entscheidbar.

Beweis. Es ist

$$\{w\$w \mid w \in L(TM_w)\} \subseteq U$$

eine Teilmenge von U, die mit der nicht entscheidbaren Sprache *STML* identifiziert werden kann. Wie oben kann man wieder zeigen, dass deswegen U nicht entscheidbar ist. □

Satz 9.8. Die Sprache U ist semi-entscheidbar.

Beweis. Wird x von TM_w akzeptiert, dann kann dies auch von der universellen *UTM* nach endlich vielen Schritten festgestellt werden. Somit kann auch $w\$x$ akzeptiert werden. □

9.4 Die nicht berechenbare fleißige Biber Funktion

Der ungarische, in den USA arbeitende Mathematiker Tibor Rado dachte sich 1962 das *busy-beaver*-Problem aus. Es kann in Kurzfassung wie folgt beschrieben werden:

- Notiere alle Turing-Maschinen mit $B = \{\#, |\}$ die genau einen Endzustand und n weitere Zustände besitzen.
- Sondere alle nicht haltenden Maschinen aus. Die verbleibenden Maschinen sind die *Biber (beaver)* mit n Zuständen.
- Starte die haltenden Maschinen auf dem leeren Arbeitsband und notiere die Anzahl der Striche.
- Bilde $bb(n)$ =Maximum der Anzahl der Striche.
- Jede Maschine mit n Zuständen, die maximal $(bb(n))$ viele Striche schreibt, heißt *fleißiger Biber (busy beaver).*

Die Funktion $bb(n)$ ist wohldefiniert, aber nicht berechenbar (siehe Satz 9.9). Wir untersuchen die Turing-Maschinen mit $n = 2$. Wir markieren in der letzten Spalte, ob die Turing-Maschine nicht stoppt (also unendlich lange läuft) mit ∞. Wenn die Turing-Maschine stoppt geben wir die Anzahl der erzeugten Striche an.

```
   1          1
###### → ###|## →          →          → ...∞

   1          2          1
###### → ###|## → ###||# →          → ...∞

   1          2          2
###### → ###|## → ###||# →          → ...∞

   1          2          1          1
###### → ###|## → ###||# → ###||# → ...∞

   1          2          1          2
###### → ###|## → ###||# → ###||# → stoppt, 2
```

In der letzten Zeile haben wir einen Biber gefunden der zwei Striche erzeugt. Wir können folgende Überführungsfunktion angeben.

<table>
<tr><th>δ</th><th>#</th><th>|</th></tr>
<tr><td>1</td><td>(2,|,r)</td><td>(2,|,r)</td></tr>
<tr><td>2</td><td>(1,|,l)</td><td>(0,|,h)</td></tr>
</table>

Weitere Suche ergibt einen Biber, der drei Striche erzeugt

```
   1          2          1          1          2
###### → ###|## → ###||# → ###||# → ##|||# → stoppt, 3
```

mit folgender Überführungsfunktion.

<table>
<tr><th>δ</th><th>#</th><th>|</th></tr>
<tr><td>1</td><td>(2,|,r)</td><td>(1,|,l)</td></tr>
<tr><td>2</td><td>(1,|,l)</td><td>(0,|,h)</td></tr>
</table>

Schließlich wird einer gefunden der vier Striche erzeugt. Dieser ist dann „fleißig“.

```
   1          2          1          2          1          2
###### → ###|## → ###||# → ###||# → ##|||# → #||||# → stoppt, 4
```

<table>
<tr><th>δ</th><th>#</th><th>|</th></tr>
<tr><td>1</td><td>(2,|,r)</td><td>(2,|,l)</td></tr>
<tr><td>2</td><td>(1,|,l)</td><td>(0,|,h)</td></tr>
</table>

Größere Biber können schnell viel mehr Striche erzeugen. Dieser Biber für $n = 6$

<table>
<tr><th>δ</th><th>#</th><th>|</th></tr>
<tr><td>1</td><td>(2,|, r)</td><td>(1,|, r)</td></tr>
<tr><td>2</td><td>(3,|, l)</td><td>(2,|, l)</td></tr>
<tr><td>3</td><td>(6,#,r)</td><td>(4,|, l)</td></tr>
<tr><td>4</td><td>(1,|, r)</td><td>(5,#,l)</td></tr>
<tr><td>5</td><td>(0,|, h)</td><td>(6,|, l)</td></tr>
<tr><td>6</td><td>(1,#,l)</td><td>(3,#,l)</td></tr>
</table>

produziert 95.524.079 Striche in 8.690.333.381.690.951 Schritten und ist *nicht* fleißig.

Sei $S(n)$ die maximale Zahl der Schritte unter allen fleißigen Bibern, die benötigt werden, um $bb(n)$ Striche zu schreiben. Folgende Ergebnisse[2] sind für fleißige Biber bekannt

n	$bb(n)$	$S(n)$	Quelle
1	1	1	Lin and Rado
2	4	6	Lin and Rado
3	6	21	Lin and Rado
4	13	107	Brady
5	≥ 4098	$\geq 47.176.870$	Marxen and Buntrock
6	$> 3{,}514 \cdot 10^{18267}$	$> 7{,}412 \cdot 10^{36534}$	Pavel Kropitz

Satz 9.9. Die Funktion $bb(n)$ ist nicht berechenbar.

Beweis. Wir werden nachweisen, dass $bb(n)$ für ein $n > n_0$ größer als jede beliebige berechenbare Funktion ist. Sei also f eine beliebige berechenbare Funktion. Wir definieren

$$\begin{aligned} F(n) &= \sum_{i=0}^{n} (f(i) + i^2) \\ &= f(0) + f(1) + 1 + \cdots + f(n) + n^2 \end{aligned}$$

Dann ist auch F berechenbar durch eine Turing-Maschine M_F. Diese habe m Zustände.

Wir betrachten nun eine Turing-Maschine M, die n Striche auf das zunächst leere Band schreibt und dann auf dem letzten Strich stehen bleibt. Das schafft sie mit n Zuständen. Dahinter schalten wir M_F, die dann $F(n)$ Striche auf das Band schreibt. Das kostet noch einmal m Zustände. Wir setzen M_F ein weiteres Mal auf das Band an, auf dem $F(n)$ Striche stehen. So erhalten wir ein Band mit insgesamt $F(F(n))$ Strichen. Diese Striche wurden von einer Maschine mit $n + 2m$ Zuständen geschrieben.

Ein fleißiger Biber mit $n + 2m$ Zuständen wird mindestens so viele Striche schreiben wie unsere Maschine. Also gilt

$$bb(n + 2m) \geq F(F(n))$$

Nun ist nach Definition $F(n) \geq n^2$, außerdem gibt es ein n_0 mit $n^2 > n + 2m$ für $n > n_0$. Also ist $F(n) > n + 2m$. Da $F(n)$ monoton ist, also für alle $x > y$ ist $F(x) > F(y)$, gilt

$$F(F(n)) > F(n + 2m) \quad .$$

Da nach Definition $F(n) > f(n)$ gilt erhalten wir insgesamt

$$bb(n + 2m) > f(n + 2m) \text{ für } n > n_0 \quad .$$

Da $f(n)$ eine beliebige berechenbar Funktion ist, kann $bb(n)$ also nicht berechenbar sein. □

[2] `http://www2.informatik.uni-stuttgart.de/fmi/ti/projects/beaver/bbb.html` `http://www.drb.insel.de/~heiner/BB`

Kapitel 10

Nicht handhabbare Probleme

Manche Probleme, wie zum Beispiel das Halteproblem, lassen sich prinzipiell nicht berechnen. Daneben gibt es allerdings auch prinzipiell berechenbare Probleme für die es praktisch nicht möglich ist sie zu lösen. Die Lösung des Problems dauert zu lange, unabhängig davon wie viel schneller und größer die Computer werden.

10.1 Laufzeit, Komplexität und Problemgröße

Definition 10.1. Die *Laufzeit* eines Algorithmus ist die Anzahl von Einzelschritten die der Algorithmus ausführt. Die Laufzeit wird meist mit einer Funktion $T(n)$ in Abhängigkeit der Eingabegröße n angegeben.

Bei der Untersuchung der Laufzeit von Algorithmen genügt meist eine Aussage darüber wie stark die Laufzeit in Abhängigkeit der Eingabegröße wächst. Dazu verwenden wir die O-Notation, die Funktionen, die gleich stark wachsen in eine gemeinsame Klasse zusammenführt. Für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ definieren wir

$$O(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c, n_0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \ .$$

und können so die Laufzeitfunktion durch eine asymptotische obere Schranke begrenzen. Praktisch bedeutet dies, dass wir für Polynome alle konstanten Faktoren ignorieren und nur den Grad des Polynoms angeben. Die *Komplexität* eines Algorithmus ist $O(f(n))$ wenn die Laufzeit $T(n)$ in $O(f(n))$ ist. Man gibt meist enge obere Schranken an.

Definition 10.2. Ein Algorithmus ist *polynomiell* wenn seine Laufzeit $T(n)$ in $O(n^k)$ für eine Konstante k ist. Ein Algorithmus ist *exponentiell* wenn er nicht polynomiell ist und wenn seine Laufzeit $T(n)$ in $O(k^{p(n)})$ für eine Konstante k und ein Polynom p ist.

Anbei einige typische Werte für Laufzeiten von Algorithmen unterschiedlicher Komplexität.

n	$O(1)$	$O(log_2(n))$	$O(n)$	$O(n \cdot log_2(n))$	$O(n^2)$	$O(n^3)$	$O(2^n)$
Grösse	Konst.	Log.	Linear	—	Quadr.	Kubisch	Exponentiell
2	1	1	2	2	4	8	4
8	1	3	8	24	64	512	256
64	1	6	64	384	4.096	262.144	10^{19}
256	1	8	256	2.048	65.536	16.777.216	10^{77}
1024	1	10	1.024	10.240	1.048.576	1.073.741.824	10^{308}
4086	1	11	4.086	49.017	16.695.396	10^{10}	$10^{1.230}$
16384	1	14	16.384	229.376	268.435.456	10^{12}	$10^{4.932}$
65536	1	16	65.536	1.048.576	4.294.967.296	10^{14}	$10^{19.728}$
262144	1	18	262.144	4.718.592	10^{10}	10^{16}	$10^{78.913}$
1048576	1	20	1.048.576	20.971.520	10^{12}	10^{18}	$10^{315.652}$

Es besteht allgemein Übereinkunft darüber, dass Probleme für die nur exponentielle Lösungsalgorithmen existieren in der Praxis nicht berechenbar – also nicht handhabbar – sind. Dagegen werden Probleme für die polynomielle Lösungsalgorithmen existieren allgemein als praktisch berechenbar angesehen.

Um Algorithmen vergleichen zu können, muss man sich auf eine klare Definition der Laufzeit einigen. Eine Möglichkeit ist eine Turing-Maschine zu Grunde zu legen und jeden Schritt der Turing-Maschine als einen Einzelschritt zu zählen. Eine Turing-Maschine braucht aber im allgemeinen mehr Schritte zur Ausführung eines beliebigen Algorithmus als ein klassischer Computer.

Satz 10.1. Eine deterministische Mehrband-Turing-Maschine kann p Schritte eines „klassischen" Computers in $O(p^3)$ Schritten simulieren, eine deterministische Einband-Turing-Maschine benötigt dafür $O(p^6)$ Schritte.

Daher hat jeder polynomielle Algorithmus auf einem Computer auch eine polynomielle Laufzeit auf einer Turing-Maschine. Das Berechnungsmodell hat keinen Einfluss auf die Aussage, ob ein Algorithmus polynomiell oder exponentiell ist. Allerdings ist der Grad des Polynoms das die Laufzeit beschreibt bei Turing-Maschinen meist höher.

Bei unterschiedlichen Berechnungsmodellen und Algorithmen muss man auch die *Problemgröße* oder *Eingabegröße n* beachten. Die Kodierung eines Problems für einen Algorithmus oder Berechnungsmodell darf sich in der Größe nur maximal polynomiell von der Kodierung für einen anderen Algorithmus oder Berechnungsmodell unterscheiden. Dies kann dadurch erreicht werden, dass man als Transformationsalgorithmus nur einen polynomiellen Algorithmus zulässt. Kodierungen von Problemstellungen die sich in polynomieller Zeit ineinander überführen lassen werden für Komplexitätsbetrachtungen als gleichwertig angesehen.

Bei Zahlen setzt man üblicherweise eine binäre Kodierung voraus. Eine Umkodierung ins Dezimalsystem oder zu einer anderen Basis ist in polynomieller Zeit möglich. Eine Darstellung einer Zahl n als n Striche auf einem Band ist allerdings *nicht* in polynomieller Zeit möglich. Zum Beispiel wären für eine natürliche Zahl x die $n = \lceil \log_2(x+1) \rceil$ Bits benötigt mindestens $2^n - 1$ Striche nötig.

10.2 Die Problemklassen *P* und *NP*

Definition 10.3. *P* ist die Menge von Problemen für die es eine deterministische Turing-Maschine gibt, die einen Lösungsalgorithmus implementiert dessen Laufzeit $T(n)$ *polynomiell* ist bezüglich der Eingabegröße n.

Wegen Satz 10.1 wissen wir, dass auch alle Probleme, die einen polynomiellen Lösungsalgorithmus auf einem beliebigen Rechenmodell eines klassischen Computers haben in *P* sind. Um von einem Problem zu zeigen, dass es in *P* ist genügt die Angabe eines polynomiellen Lösungsalgorithmus.

Beispiele für Probleme in *P* sind viele praktische Probleme wie Sortieren (MergeSort), Finden eines minimalen spannenden Baums in einem Graphen (Kruskal-Algorithmus), und so weiter.

Meist beschränkt man sich bei Problemen auf *Entscheidungsprobleme* bei der die Ausgabe nur ja oder nein ist. Man kann zeigen, dass zugehörige Entscheidungsprobleme genauso schwierig sind wie verwandte allgemeine Berechnungsprobleme.

Definition 10.4. *NP* ist die Menge von Problemen für die es eine *nicht*-deterministische Turing-Maschine gibt, die einen Lösungsalgorithmus implementiert dessen Laufzeit $T(n)$ *polynomiell* ist bezüglich der Eingabegröße n für alle möglichen Berechnungswege.

Es gibt viele praktische Probleme von denen man zeigen kann, dass sie in *NP* sind. Ein Beispiel ist das Knapsack-Problem.

Beispiel 10.1. Ein Wanderer will seinen Rucksack packen. Der Rucksack darf maximal G wiegen. Es stehen n Objekte zur Verfügung. Jedes Objekt hat ein Gewicht g_i und einen Nutzen a_i. Der Wanderer will seinen Rucksack so packen, dass er einen maximalen Nutzen hat. Bei $x_i = 1$ wird Objekt i mitgenommen, bei $x_i = 0$ nicht. Der Wanderer muss also das Optimierungsproblem

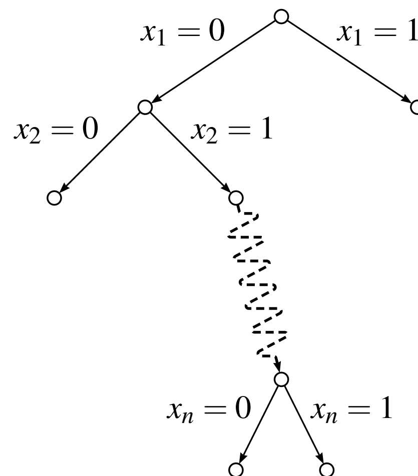
$$\begin{array}{rll} \text{Maximiere} & a_1 \cdot x_1 + \cdots + a_n \cdot x_n & \\ \text{Nebenbedingung} & g_1 \cdot x_1 + \cdots + g_n \cdot x_n & \leq G \end{array}$$

lösen. Ein zugehöriges Entscheidungsproblem ist ob es eine Bepackung des Rucksacks gibt, die einen gegebenen Nutzen A erreicht, also ob

$$\begin{array}{rcl} a_1 \cdot x_1 + \cdots + a_n \cdot x_n & \geq & A \\ g_1 \cdot x_1 + \cdots + g_n \cdot x_n & \leq & G \end{array}$$

eine Lösung besitzt.

Ein Entscheidungsalgorithmus mit einem nicht-deterministischen Rechenmodell läuft durch alle Objekte und nimmt nicht-deterministisch einmal das Objekt dazu ($x_i = 1$) und einmal nicht ($x_i = 0$).



Wenn das Objekt dazu genommen wird, dann wird der Nutzen addiert. Wenn das Gesamtgewicht überschritten wird, dann wird der Pfad nicht weiter verfolgt. Ein Berechnungspfad stoppt, sobald der Nutzen A erreicht wurde. Die Länge jedes möglichen Berechnungspfads ist linear abhängig von n. Daher ist das Knapsack-Problem in *NP*.

Satz 10.2. Jedes Problem in P ist auch in *NP*. Es gilt

$$P \subseteq NP \quad .$$

Beweis. Wenn ein Problem L in P ist, dann gibt es eine deterministische Turing-Maschine die L in polynomieller Zeit löst. Diese deterministische Turing-Maschine kann als nicht-deterministische Turing-Maschine angesehen werden, bei der jeder Schritt nur einen möglichen Folgeschritt hat. Also gibt es eine nicht-deterministische Turing-Maschine, die L in polynomieller Zeit löst. Also ist L in *NP*. □

Satz 10.3. Für jedes Problem in *NP* gibt es einen exponentiellen Lösungsalgorithmus.

Ein exponentieller Lösungsalgorithmus für Probleme in *NP* kann zum Beispiel die Ausführung aller nicht-deterministischen Lösungswege simulieren. Der Baum aller möglichen Berechnungswege kann mit einer Breitensuche abgelaufen werden. Wenn die Anzahl der möglichen Verzweigungen durch k und der längste Berechnungsweg durch das Polynom $p(n)$ begrenzt ist, dann ist die Komplexität des Lösungsalgorithmus $O(k^{p(n)})$.

Die Probleme in *NP* haben die Eigenschaft, dass eine richtige Lösung in polynomieller Zeit *geraten* werden kann. Das vollständige Raten (vollständige Aufzählung) entspricht

dem Nicht-Determinismus. Man kann daher auch Lösungskandidaten für Probleme in *NP* in polynomieller Zeit verifizieren. Zum Beispiel ist bei dem Knapsack-Problem der Test ob eine gegebene Auswahl an Gegenständen weniger als das Gewicht G und mindestens den Nutzen A hat in linearer Zeit ausführbar.

Da viele wichtige praktische Probleme in *NP* sind stellt sich die Frage, ob es polynomielle Lösungsalgorithmen gibt für alle Probleme in *NP*. Falls ja, dann wären auch alle Probleme in *NP* auch in *P*. Obwohl schon viel Gehirnschmalz in diese Frage geflossen ist, ist dieses Problem bis heute noch offen.

Das wohl zurzeit größte offene Problem der theoretischen Informatik ist der Beweis für

$$P \neq NP \quad .$$

Wie schon die Formulierung zeigt, geht man heute allgemein davon aus, dass es Probleme in *NP* gibt, für die es keinen polynomiellen Lösungsalgorithmus gibt, dass also $P \subset NP$ gilt.

Zu beweisen, dass es keinen Algorithmus geben kann der eine gewisse Eigenschaft hat – in dem Fall polynomiell – ist immer ein sehr schwieriges Unterfangen.

10.3 *NP*-vollständige Probleme

Auch wenn der Beweis für $P \neq NP$ bisher noch nicht gelungen ist wurde eine Klasse von Problemen definiert, die mindestens so komplex sind wie jedes andere Problem in *NP*.

Definition 10.5. Ein Problem L_1 heißt *polynomiell auf L_2 reduzierbar*, wenn es eine Transformation (Abbildung) aller Instanzen von L_1 auf Instanzen von L_2 gibt, die mit polynomieller Komplexität berechenbar ist.

Definition 10.6. Ein Problem L heißt *NP*-vollständig, wenn

- L in *NP* ist und
- jedes Problem $L' \in NP$ polynomiell auf L reduzierbar ist.

Falls irgendwann jemand einen polynomiellen Lösungsalgorithmus für ein *NP*-vollständiges Problem L finden würde, dann hätte man einen polynomiellen Lösungsalgorithmus für jedes Problem in *NP*. Es würde dann gelten, dass $P = NP$. Man würde dann dazu einfach irgendein beliebiges Problem L' aus *NP* auf das Problem L reduzieren und danach das Problem L mit dem neuen polynomiellen Algorithmus lösen. Da sowohl der Transformationsalgorithmus als auch der Lösungsalgorithmus polynomiell wären, hätte man einen polynomiellen Lösungsalgorithmus für jedes Problem L' in *NP*.

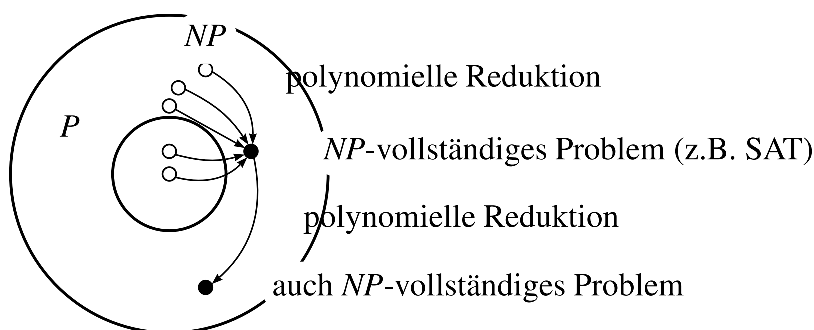
Obwohl man bisher noch nicht nachweisen konnte, dass es keinen polynomiellen Lösungsalgorithmus für *NP*-vollständige Probleme gibt, konnte man für eine Reihe wichtiger Problem zeigen, dass sie *NP*-vollständig sind. Das bekannteste Beispiel ist das Erfüllbarkeitsproblem.

Eine aussagenlogische Formel besteht aus einem Ausdruck über den Konstanten wahr ($\top$) und falsch ($\bot$), den boolesche Variablen x_i und den Operationen Und ($\wedge$), Oder ($\vee$) und Nicht ($\neg$, oder $\bar{}$). Das aussagenlogische *Erfüllbarkeitsproblem* (SAT) ist ein Entscheidungsproblem das wahr ergibt, wenn für eine aussagenlogische Formel eine Variablenbelegung existiert, so dass die aussagenlogische Formel zu wahr ($\top$) ausgewertet wird.

Satz 10.4. (Cook, 1971) Das Erfüllbarkeitsproblem für aussagenlogische Formeln (SAT) ist *NP*-vollständig.

Beweis. (Beweisidee) Um den Satz zu zeigen, wird eine Sprache L einer beliebigen nichtdeterministisch polynomiellen Turing-Maschine betrachtet. Jedes Problem *NP* kann als Sprache L einer solchen Turing-Maschine kodiert werden. Es wird dann ein polynomieller Transformationsalgorithmus angegeben, der für jedes Wort von L eine aussagenlogische Formel konstruiert, die genau dann erfüllbar ist, wenn das Wort in L ist. □

Wenn man mit SAT ein erstes Beispiel für ein *NP*-vollständiges Problem gefunden hat, genügt es für den Beweis, dass andere Probleme *NP*-vollständig sind eine polynomielle Reduktion von SAT auf das andere Problem zu finden (und nicht von dem neuen Problem auf SAT).



Es wurde schon für eine Vielzahl weiterer Probleme gezeigt, dass sie *NP*-vollständig sind. Darunter zum Beispiel das Knapsack-Problem, das Problem des Handlungsreisenden (Traveling Salesman Problem, TSP), ganzzahlige mathematische Optimierung, und so weiter. Auch für einige vereinfachte SAT-Probleme wurde gezeigt, dass diese *NP*-vollständig sind. Dazu zählen

- KSAT: SAT-Problem für Formeln in *Konjunktiver NormalForm (KNF)*. Also aussagenlogische Ausdrücke der Form

$$\begin{array}{rl} (l_{1_1} \vee \ldots \vee l_{1_n}) & \wedge \\ \ldots & \wedge \\ (l_{m_1} \vee \ldots \vee l_{m_n}) & \end{array}$$

  wobei l_{j_k} *Literale* sind, also $l_{j_k} = x_i$ oder $l_{j_k} = \overline{x_i}$. Ein Ausdruck $l_{j_1} \vee \ldots \vee l_{j_n}$ heißt auch *Klausel*.

- 3SAT: SAT-Problem für Formeln in konjunktiver Normalform mit drei Literalen, also aussagenlogische Ausdrücke der Form

$$\begin{array}{rl} (l_{1_1} \vee l_{1_2} \vee l_{1_3}) & \wedge \\ \ldots & \wedge \\ (l_{m_1} \vee l_{m_2} \vee l_{m_3}) & \end{array}$$

Um zu zeigen, dass ein Problem *NP*-vollständig ist wird häufig 3SAT auf das Problem reduziert.

Es würde reichen für eines dieser *NP*-vollständigen Probleme einen polynomiellen Lösungsalgorithmus zu finden, um einen polynomiellen Lösungsalgorithmus für *alle* Probleme in *NP* zu finden. Der jahrzehntelange erfolglose Versuch vieler Wissenschaftler für eines dieser Probleme einen solchen polynomiellen Lösungsalgorithmus zu finden wird als weiteres Indiz für $P \neq NP$ angesehen.

Für manche Probleme kann man zwar zeigen, dass man damit jedes Problem in *NP* lösen kann, aber es ist nicht gelungen für diese Probleme zu zeigen, dass sie in *NP* sind.

Definition 10.7. Ein Problem L heißt *NP*-hart, wenn jedes Problem $L' \in NP$ polynomiell auf L reduzierbar ist.

NP-harte Probleme sind also mindestens so komplex wie *NP*-vollständige Probleme. Ein Beweis, dass ein *NP*-hartes Problem nur einen exponentiellen Lösungsalgorithmus besitzt hilft aber nicht weiter in der Frage ob $P \neq NP$.