Inhaltsverzeichnis

Kapitel 1

Formale Sprachen

Eine natürliche Sprache wird in erster Linie durch sprachliche Sätze charakterisiert, die durch Menschen artikuliert werden. Dabei setzt man grammatikalische Korrektheit und sinnvoller Gebrauch voraus. Natürliche Sprachen sind komplex und können nicht vollständig durch ein allgemein gültiges Regelwerk beschrieben werden. Zwar verfügt man heute über sehr gute Regelwerke für die grammatikalische Korrektheit — dies nennt man **Syntax** — von Sätzen, die Bedeutung — dies nennt man die **Semantik** — eines Satzes oder eines Textes ist nur ansatzweise formal erfassbar.

Ein großes Problem natürlicher Sprachen ist unter anderem, dass Mehrdeutigkeiten in den Äußerungen bestehen. Betrachtet man die Anweisung

> *Öffne die Datei mit dem Editor*,

dann kann man den Satzteil *mit dem Editor* als Attribut des Objekts *Datei* verstehen, oder als adverbiale Bestimmung des Verbs *öffnen*.

Aus diesem Grund sind natürliche Sprachen nicht geeignet, Aufgaben zu beschreiben, die ein Computer lösen soll.[1] Besser für diesen Zweck geeignet sind **formale Sprachen** wie Programmiersprachen, in denen auf der Grundlage einer relativ kleinen Menge von syntaktischen Regeln Programme formuliert werden, die eine Semantik aufweisen, die von Rechnern verstanden wird.

Die Theorie der formalen Sprachen bietet eine theoretische Grundlage für alle Sprachen an, die mit automatischen Verfahren verarbeitet werden. Hierbei werden insbesondere syntaktische Aspekte betrachtet. Zentrale Begriffe wie Alphabet, Buchstaben oder Worte, die man bei dem Aufbau natürlicher Sprachen verwendet, werden dabei generalisiert.

[1] Außer in sehr eingeschränkter Form.

1.1 Buchstaben, Wörter und Sprachen

Zum Verständnis der folgenden Definitionen ist die Veranschaulichung anhand der analogen Verwendung in natürlichen Sprachen sehr hilfreich.

Definition [1.1]:

Ein **Alphabet** Σ ist eine endliche, nichtleere Menge von **Zeichen**, zusammen mit einer auf Σ definierten totalen Ordnung.

Anmerkungen:

1. Die Zeichen eines Alphabets nennt man auch **Buchstaben** oder Symbole.
2. Totale Ordnung bedeutet, dass auf der Menge Σ eine Ordnungsrelation erklärt ist, so dass für jedes Element $a_i \in \Sigma$ gilt:

$$a_1 << a_2 << \cdots << a_i << \cdots << a_n.$$

Definition [1.2]:

Sei Σ ein Alphabet. Endliche Folgen

$$(a_1 a_2 \dots a_k)$$

mit $a_i \in \Sigma \; \forall \; i = 1, 2, \dots, k$ heissen **Wörter** über dem Alphabet Σ mit der Länge k. Es gibt ein Wort der Länge 0, dieses Wort bezeichnet man mit λ und nennt es das **leere Wort**.

Die **Länge** eines Worts w bezeichnen wir mit $\mid w \mid$.

Anmerkung:

Der Übersichtlichkeit halber verzichtet man in den meisten Fällen bei der Bildung von Worten auf die Klammern um die Zeichenfolgen, wie in der Definition [1.2] vorgegeben. Dies ist jedoch nicht immer möglich.[2]

Definition [1.3]:

Wir bezeichnen mit Σ^k die Menge aller Worte der Länge $k \in \mathbb{N}$ über dem Alphabet Σ. Die Menge aller Worte über Σ bezeichnet man mit Σ^*. Die Menge aller nichtleeren Worte über dem Alphabet Σ bezeichnet man mit Σ^+.

[2] Betrachtet man beispielsweise das Alphabet $\Sigma = \{x, xx\}$, dann ist nicht eindeutig erkennbar, aus welchen Zeichen das Wort $w = xxx$ gebildet ist.

Anmerkung:

Die Operation * nennt man auch KLEENE Stern Operation.

Definition [1.4]:

Jede Teilmenge von Σ^* heisst **Sprache**.

In diesem allgemeinen Sinn wird jede Menge als Alphabet aufgefasst, vorausgesetzt die Menge ist endlich und nichtleer. Jede Wortmenge wird als Sprache bezeichnet.

Bei natürlichen Sprachen unterscheidet man zwei Betrachtungsebenen. Auf der untersten Ebene werden Wörter über dem Alphabet $\{a, b, c, \ldots, z\}$ gebildet, aus denen der Wortschatz der natürlichen Sprache gebildet wird.[3] Auf einer höheren Ebene dient dieser Wortschatz wieder selbst als Alphabet, die Worte, die daraus gebildet werden sind die grammatikalisch korrekten und semantisch zulässigen Sätze. Diese beiden Aspekte finden sich analog bei Programmiersprachen, die auf der Theorie der formalen Sprachen basieren.

Die wichtigste Operation mit Wörtern ist die Verkettung einzelner Wörter. Diese Operation nennt man auch **Konkatenation**.

Definition [1.5]:

Sei Σ ein Alphabet. Die Konkatenation von Wörtern $w_1, w_2 \in \Sigma^*$ ist:

$$w_1 \circ w_2 = w_1 w_2 = \underbrace{x_1 x_2 \ldots x_k}_{x} \underbrace{y_1 y_2 \ldots y_l}_{y}$$

mit $x_i \in \Sigma, y_j \in \Sigma$. Für die n–fache Konkatenation eines Wortess $w \in \Sigma^*$ mit sich selbst schreiben wir:

$$w^n = \underbrace{www \ldots wx}_{n \text{ mal}}, \quad w \in \Sigma^*.$$

Insbesondere ist

$$x^0 = \lambda,$$

wobei das leere Wort λ das neutrale Element der Konkatenationsoperation bildet:

$$w\lambda = \lambda w = w \quad \forall\, w \in \Sigma^*.$$

[3] Im Deutschen ist dies die Menge aller Wörter, die im Duden eingetragen sind, im Englischen sind dies die Worte im Dictionary.

Anmerkung:

Das Tripel $(\Sigma^*, \circ, \lambda)$ hat die algebraische Struktur eines Monoids, mit neutralem Element λ.

Beispiel [1.1]

Betrachte das binäre Alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. Die Wörter über dem binären Alphabet $\{\mathtt{0}, \mathtt{1}\}$ sind alle endlichen Bitstrings inklusive dem leeren Wort, *i.e.*

$$\Sigma^* = \{\lambda, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{11}, \mathtt{000}, \mathtt{001}, \mathtt{010}, \ldots\}.$$

Eine Sprache über dem binären Alphabet ist eine Teilmenge von Σ^*, beispielsweise alle vorzeichenlosen Binärzahlen ohne führende 0en:

$$L = \{\mathtt{0}, \mathtt{1}, \mathtt{10}, \mathtt{11}, \mathtt{100}, \mathtt{101}, \mathtt{110}, \mathtt{111}, \mathtt{1000}, \ldots\} \subset \{\mathtt{0}, \mathtt{1}\}^*.$$

Ein weiteres Beispiel ist die Menge aller Worte über $\{\mathtt{0}, \mathtt{1}\}$ der Länge 8, *i.e.* die Bytes:

$$\begin{aligned} L &= \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid\mid w \mid = 8\} \\ &= \{\mathtt{0000\,0000}, \mathtt{0000\,0001}, \ldots, \mathtt{1111\,1111}\}. \end{aligned}$$

Beispiel [1.2]

Sei

$$\Sigma = \{0, 1, 2, \ldots, 9\}$$

das dezimale Alphabet. Dieses Alphabet ist geeignet zur Darstellung von natürlichen Zahlen in Dezimalschreibweise, z.B. 32198.

Beispiel [1.3]

Betrachte das unäre Alphabet

$$\Sigma = \{|\}.$$

Dieses einelementige Alphabet wird gelegentlich verwendet, um natürliche Zahlen zu codieren, *e.g.*

$$\begin{aligned} 0 &\longrightarrow \ | \\ 1 &\longrightarrow \ || \\ 2 &\longrightarrow \ ||| \\ &\text{usw.} \end{aligned}$$

Damit erhält man eine Darstellung der natürlichen Zahlen der Form

$$n \in \mathbb{N} \longrightarrow |^{n+1} = \underbrace{| \, | \ \cdots \ |}_{n+1 \text{ mal}} .$$

Beispiel [1.4]

Betrachte das Alphabet der lateinischen Schrift

$$\Sigma = \{a, b, c, \ldots, z\}.$$

Dies ist das den Wörtern von natürlichen Sprachen zugrundeliegende Alphabet, je nach Länderspezifikation erweitert durch Großbuchstaben, Umlaute und/oder Sonderzeichen.

Beispiel [1.5]

Betrachte

$$\Sigma = \text{ Menge der Einträge eines Lexikons.}$$

Beispielsweise bildet die Menge der Einträge im Duden ein Alphabet. Mit diesem Alphabet lassen sich natürlichsprachliche Äußerungen wie

Das Haus ist aus Beton gebaut

formulieren. Solche korrekt gebildeten Sätze — auf dieser Ebene sind diese Sätze Wörter über dem Alphabet der Dudeneinträge — stellen die deutsche Sprache dar. Ob ein Satz korrekt ist kann über Grammatikregeln entschieden werden.

Beispiel [1.6]

Betrachte die beiden Alphabete

$$\Sigma = \textbf{ASCII}^4$$

oder

$$\Sigma = \textbf{EBCDIC}^5.$$

Programme einer Programmiersprache wie C, Java etc. werden üblicherweise mit einem dieser Alphabete formuliert (Quellcode). Man kann nun Programme als endliche Folgen solcher Zeichen auffassen, zweckmäßiger ist jedoch wie in natürlichen Sprachen eine weitere Ebene zu betrachten.
Auf der unteren Ebene werden mit Hilfe von ASCII– oder EBCDIC–Zeichen beim Erstellen eines Quellcodes sogenannte Eingabesymbole gebildet.[6] Bei der Programmiersprache C differenziert man fünf Arten von Eingabesymbolen:

- Schlüsselworte
- Bezeichner
- Literale
- Operatoren
- Begrenzer

Der erste Schritt bei der Programmübersetzung, *i.e.* Kompilierung, besteht darin, die Eingabesymbole zu identifizieren. Beispielsweise wird `while` als Schlüsselwort identifiziert, dagegen ist `whilexx` ein Bezeichner.

Beispiel [1.7]

$$\Sigma = \text{Menge der Eingabesymbole einer Programmiersprache}$$

Auf dieser Stufe sind Programme Wörter von Eingabesymbolen. Als Programmiersprache kann man die Menge aller endlichen Folgen von Eingabesymbolen auffassen, die ein korrektes Programm darstellen. Hierbei wird mit Hilfe von Grammatiken festgelegt, wie die einzelnen Eingabesymbole zu einem korrekten Programm zusammengefügt werden.

Betrachte die folgende C Anweisung:

```
while (x3 < 100) x3+1.
```

In dieser Interpretation stellt diese Anweisung ein Teilwort eines Programms dar, bestehend aus

[4] ASCII = American Standard Code for Information Interchange.
[5] EBCDIC = Extended Binary Coded Decimal Interchange Code
[6] Diese Eingabesymbole nennt man auch *Token*.

- `while` Schlüsselwort
- `(` Begrenzer
- `x3` Bezeichner
- $<$ Operator
- 100 Literal
- `)` Begrenzer
- `x3` Bezeichner
- $+$ Operator
- 1 Literal

$\Longrightarrow$ Übungen [1.6], [1.7], [1.8]

1.2 Klassen von unendlichen Sprachen

Die Menge Σ^* aller Wörter über einem Alphabet Σ ist immer unendlich. Sie ist daher unabhängig von dem zugrundliegende Alphabet eine unendliche Sprache. Mit unendlichen Sprachen wird man in der Informatik häufig konfrontiert, daher ist es zweckmäßig, genauer zu untersuchen, welche Rolle das Problem der Unendlichkeit bei formalen Sprachen spielt.

Aus der Mathematik — insbesondere der Mengenlehre — ist bekannt, dass man formal mit unendlichen Zahlenmegen umgehen kann. Man unterscheidet prinzipiel zwei Kategorien:

- **abzählbar unendliche Mengen**
- **überabzählbar unendliche Mengen**

1.2.1 Abzählbare Sprachen

Die Eigenschaft der Abzählbarkeit einer Menge drückt im Wesentlichen aus, dass man von dem ersten Element, zweiten Element, usw. sprechen kann. Bei abzählbaren Mengen muss es also möglich sein, die Zahlen $1, 2, 3, 4, \ldots$ auf die Elemente der Menge so zu verteilen, dass *jedes* Element der Menge genau eine Nummer erhält. Formal kann man dies folgendermaßen formulieren.

Definition [1.6]:

Eine Menge M nennt man **abzählbar**,[7], wenn es eine bijektive Funktion

$$f : \mathbb{N} \longrightarrow M$$

gibt, oder falls $M = \emptyset$.

Ist eine Menge nicht abzählbar, dann nennt man diese Menge **überabzählbar**.

Anmerkungen:

1. Sprachen sind erst mal Mengen und können daher als abzählbar charakterisiert werden. Wir werden sehen, dass alle Sprachen abzählbar sind. Dies resultiert aus der Forderung, dass die zugrunde liegende Alphabete endliche Mengen sind.

2. In der Mengenlehre legt man fest, dass eine Menge M die gleiche **Kardinalität** hat wie $\mathbb{N}$, wenn man eine Bijektion f von $\mathbb{N}$ nach M angeben kann. Die Kardinalität der Menge der natürlichen Zahlen bezeichnet man mit $\aleph_0$, *i.e.*

$$| \mathbb{N} | = \aleph_0.$$

Die Bezeichnung $\aleph_0$ steht also für *abzählbar unendlich*.

[7] Synonym dazu verwendet man auch den Begriff *abzählbar unendlich*

Beispiel [1.8]

Die Menge der Zweierpotenzen

$$M = \{1, 2, 4, 8, 16, 32, \ldots\}$$

ist abzählbar. Dieser Sachverhalt wird aus der folgenden Auflistung ersichtlich:

$$\begin{array}{lcr} \mathbb{N} & & M \\ 0 & \longrightarrow & 1 = 2^0 \\ 1 & \longrightarrow & 2 = 2^1 \\ 2 & \longrightarrow & 4 = 2^2 \\ 3 & \longrightarrow & 8 = 2^3 \\ 4 & \longrightarrow & 16 = 2^4 \\ & \vdots & \end{array}$$

Formal haben wir damit eine Abbildung

$$f \,:\, \mathbb{N} \longrightarrow M$$

mit der Zuordnung

$$n \longmapsto f(n) = 2^n.$$

Dies ist eine surjektive Abbildung, denn *jede* Zweierpotenz 2^n hat ein Urbild, nämlich n. Die Abbildung ist sogar eineindeutig, *i.e.* injektiv. Die Umkehrabbildung — diese sagt aus, welches Element aus M welcher Nummer zugeordnet ist — ist

$$f^{-1}(m) = \log_2 m = n.$$

Beispiel [1.9]

Die Menge der ganzen Zahlen $\mathbb{Z}$ ist abzählbar.

Beweis:

Wir betrachten die Menge

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3 \ldots\}$$

und definieren eine Abbildung

$$f \,:\, \mathbb{Z} \longrightarrow \mathbb{N} \tag{1.1a}$$

durch

$$f(x) = \begin{cases} 2x & \text{falls } x \geq 0, \\ -2x - 1 & \text{falls } x < 0 \end{cases}. \tag{1.1b}$$

Damit ist

$$f(0) = 0, f(1) = 2, f(2) = 4, f(3) = 6, f(-1) = 1, f(-2) = 3, \ldots.$$

Mit der Abbildung (1.1a) können wir also jeder ganzen Zahl eine Nummer zuordnen, beispielweise ist $x = -4$ die $n = 7$te Zahl in der Aufzählung.

Die in der Definition geforderte Bijektivität ermöglicht es nun umgekehrt festzustellen, welche ganze Zahl x die Nummer n hat. Für eine bijektive Funktion existiert stets die Umkehrfunktion, die ist für die Abbildung (1.1a), (1.1b) gegeben durch

$$g \ : \ \mathbb{N} \longrightarrow \mathbb{Z} \tag{1.1c}$$

mit

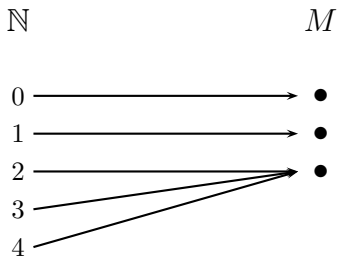
$$g(n) = \begin{cases} \frac{n}{2} & \text{falls } n \text{ gerade,} \\ -\frac{n+1}{2} & \text{falls } n \text{ ungerade.} \end{cases} . \tag{1.1d}$$

Damit haben wir:

n		$g(n)$
0	$\longleftrightarrow$	0
1	$\longleftrightarrow$	-1
2	$\longleftrightarrow$	1
3	$\longleftrightarrow$	-2
4	$\longleftrightarrow$	2
5	$\longleftrightarrow$	-3
6	$\longleftrightarrow$	3
7	$\longleftrightarrow$	-4
$\vdots$		$\vdots$

Beispiel [1.10]

Jede endliche Menge ist abzählbar. Denn unter einer surjektiven Abbildung dürfen Elemente des Bildbereichs beliebig oft als Bild auftreten (das ist bei Injektivität nicht möglich). Die folgende Zuordnung macht dies deutlich.



Wie das Beispiel [1.10] zeigt, ist die Untersuchung der Abzählbarkeit lediglich für unendliche Mengen interessant, da alle endlichen Mengen abzählbar sind.

Das folgende Beispiel betrachtet eine unendliche Zahlenmenge, bei der es nicht auf den ersten Blick ersichtlich ist, dass sie abzählbar ist.

Beispiel [1.11]

Die rationalen Zahlen $\mathbb{Q}$ werden durch die Menge der Zahlen gebildet, die sich als Bruch p/q darstellen lassen, wobei p, q ganze Zahlen sind und $q \neq 0$. Typische rationale Zahlen sind

$$\frac{2}{3}, \quad \frac{5}{9}, \quad -\frac{7}{4}, \quad \frac{2}{1}, \ldots,$$

typische Zahlen, die man nicht als Bruch darstellen kann — diese nennt man **irrationale Zahlen** – sind $\pi, e, \sqrt{2}$ usw.

Obwohl in jedem noch so kleinen Intervall der Zahlengeraden unendlich viele rationale Zahlen liegen — der Fachterminus ist, dass die rationalen Zahlen *dicht* auf der Zahlengeraden liegen — sind die rationalen Zahlen abzählbar.

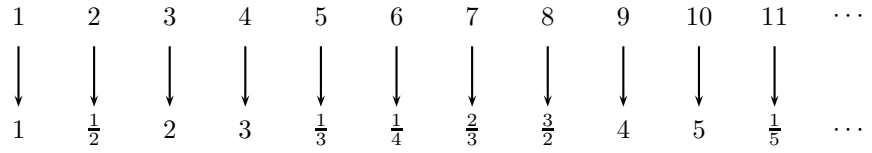
Das verwendete Verfahren nennt man **1. Cantorsches Diagonalisierungsverfahren** ,[8] benannt nach dem Mathematiker Georg Cantor (1845 – 1918).

Wir zeigen zunächst, dass die Menge der natürlichen Zahlen $\mathbb{N}$ und die Menge der positiven rationalen Zahlen $\mathbb{Q}_+$ gleich mächtig sind, das bedeutet, die Menge der positiven rationalen Zahlen ist abzählbar.

Dies lässt sich zeigen, wenn man die rationalen Zahlen auflistet wie in der Abbildung [1.1].

Dieses Schema zählt man dann diagonal ab, wobei nicht vollständig gekürzte Brüche übersprungen werden. Dies ist in der Abbildung [1.2] dargestellt.

Man erhält auf diese Weise eine Abzählung der positiven rationalen Zahlen:



[8] Siehe beispielweise Courant und Robbins, [55], Kapitel 4.2.

$$\begin{array}{cccccc}
\frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \cdots\cdots \\
\frac{2}{1} & \frac{2}{2} & \frac{2}{3} & \frac{2}{4} & \frac{2}{5} & \frac{2}{6} & \cdots\cdots \\
\frac{3}{1} & \frac{3}{2} & \frac{3}{3} & \frac{3}{4} & \frac{3}{5} & \frac{3}{6} & \cdots\cdots \\
\frac{4}{1} & \frac{4}{2} & \frac{4}{3} & \frac{4}{4} & \frac{4}{5} & \frac{4}{6} & \cdots\cdots \\
\frac{5}{1} & \frac{5}{2} & \frac{5}{3} & \frac{5}{4} & \frac{5}{5} & \frac{5}{6} & \cdots\cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots &
\end{array}$$

Abbildung 1.1: Systematische Auflistung aller positiven rationalen Zahlen.

Durch das Überspringen der kürzbaren Brüche liegt für jede positive rationale Zahl genau ein Repräsentant in dieser Abzählung. Hierdurch erhält man die notwendige Bijektion zwischen der Menge der natürlichen Zahlen $\mathbb{N}$ und der Menge der positiven rationalen Zahlen $\mathbb{Q}_+$.

Um zu zeigen, dass die Menge der natürlichen Zahlen $\mathbb{N}$ gleichmächtig zur Menge der rationalen Zahlen ist, wird diese Abzählung einfach erweitert. Wir fügen die 0 vor der Eins hinzu und schreiben hinter jeder rationalen Zahl ihr Negatives.

$$\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & \cdots \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
0 & 1 & -1 & \frac{1}{2} & -\frac{1}{2} & 2 & -2 & 3 & -3 & \frac{1}{3} & -\frac{1}{3} & \frac{1}{4} & -\frac{1}{4} & \frac{2}{3} & -\frac{2}{3} & \frac{3}{2} & \cdots
\end{array}$$

Damit erhält man eine Bijektion zwischen der Menge der natürlichen Zahlen $\mathbb{N}$ und den rationalen Zahlen $\mathbb{Q}$. Dies impliziert, dass die Menge der rationalen Zahlen abzählbar ist.

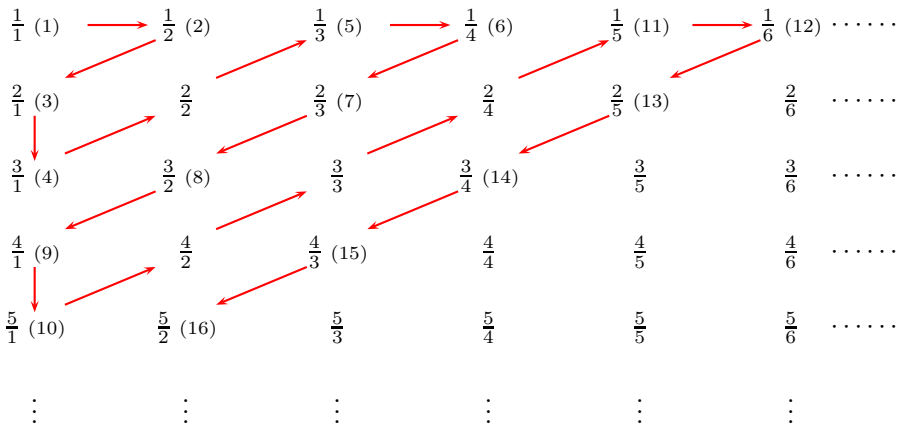


Abbildung 1.2: Aufzählverfahren für positive Brüche — 1. Diagonalisierungsverfahren von CANTOR.

1.2.2 Gödelisierung

Mit dem Begriff **Gödelisierung** bezeichnet man ein Verfahren, jedem Wort einer formalen Sprache in eineindeutiger Weise eine bestimmte Zahl zuzuordnen. Dieses Verfahren ist nach KURT GÖDEL benannt, der erstmals ein solches Verfahren angab, um seinen Unvollständigkeitssatz zu beweisen.[9]

Ausgangspunkt ist die Reihe der Primzahlen:

$$p_1 = 2, p_2 = 3, p_3 = 5, p_4 = 7, p_5 = 11, p_6 = 13, \ldots$$

Die GÖDEL Zahl der Zahlenfolge

$$s = (n_1, n_2, n_3, \ldots, n_k), \quad n_i \in \mathbb{N}, i = 1, 2, \ldots, k$$

ist die positive ganze Zahl

$$c(s) = p_1^{n_1} \cdot p_2^{n_2} \cdot \ldots \cdot p_k^{n_k} = \Pi_{i=1}^{k} p_i^{n_i}. \tag{1.2}$$

Ist beispielweise

$$s = (2, 4, 1, 0, 1),$$

dann ist die GÖDEL Zahl:

$$c(s) = 2^2 \cdot 3^4 \cdot 5^1 \cdot 7^0 \cdot 11^1 = 17\,820.$$

Da es nach dem Satz von EUKLID unendlich viele Primzahlen gibt, kann man auf die Art und Weise jede Zahl mit beliebig vielen Ziffern durch ihre GÖDEL Zahl codieren.

[9] Literatur über dieses Thema ist unter anderem HOFSTADTER [16], Kapitel IX, DAVIS *et.al.* [29] oder ROGER PENROSE [20].

Umgekehrt: Gemäß dem Fundamentalsatz der Arithmetik — dieser sagt aus, dass jede natürliche Zahl in eindeutiger Weise als Produkt von Potenzen von Primzahlen geschrieben werden kann — kann aus der GÖDEL Nummer das Wort wieder rekonstruiert werden.

Umgekehrt kann also aus einer gegebenen GÖDEL Zahl auf die Zahlenfolge geschlossen werden, wie im folgenden Beispiel.

$$c(s) = 72\,600.$$

Dann können wir die Zahl 72 600 faktorisieren und erhalten:

$$72\,600 = 2^3 \cdot 3^1 \cdot 5^2 \cdot 7^0 \cdot 11^2.$$

Dann ergibt sich die Zahlenfolge

$$s = (3, 1, 2, 0, 2).$$

Um dieses Verfahren auf Worte über einem Alphabet zu erweitern, muss lediglich eine Codierung der Zeichen des Alphabets durch Zahlen durchgeführt werden. Sei w ein Wort über dem Alphabet

$$\Sigma = \{a_1, a_2, a_3, \ldots\}.$$

Dann ist die GÖDEL Nummer des Wortes w die positive Zahl $c(w)$, wobei der Index des i–ten Zeichens von w der Exponent von p_i in der Faktorisierung von $c(w)$ ist. Beispielweise ist das Wort

$$w = a_4a_2a_3a_1$$

codiert durch

$$c(w) = 2^4 \cdot 3^2 \cdot 5^3 \cdot 7^1 = 126\,000.$$

Beispiel [1.12]

Wir betrachten zur Illustration das Alphabet $\Sigma = \{a, b, c\}$. Wir wollen nun jedem Wort über diesem Alphabet, das zu einer bestimmten Sprache L gehört, eine eineindeutige Nummer zuordnen, diese Nummer ist die **Gödel Zahl**. Die Zuordung einer Nummer zu einem Wort nennt man auch **Codierung**.

Eine Möglichkeit, dies zu tun besteht darin, jedem Buchstaben zunächst eine Nummer zuzuordnen, *e.g.* liegt der Sprache L das Alphabet $\Sigma = \{a, b, c\}$ zugrunde, setzt man: a ist die 1, b die 2 und c erhält die Nummer 3.[10] Die Gödelisierung eines Wortes über dem Alphabet Σ besteht nun darin, die dem Buchstaben entsprechenden Potenzen der fortlaufenden Primzahlen zu multiplizieren.

Betrachte das Wort $w = abaccab$. Wir setzen nun:

[10] Das ist exakt das obige Verfahren, in diesem Beispiel haben wir gesetzt

$$a \longleftrightarrow a_1, b \longleftrightarrow a_2, c \longleftrightarrow a_3.$$

- Das a an erster Stelle hat den Wert $2^1 = 2$.
- Das b an zweiter Stelle hat den Wert $3^2 = 9$.
- Das a an dritter Stelle hat den Wert $5^1 = 5$.
- Das c an vierter Stelle hat den Wert $7^3 = 343$.
- Das c an fünfter Stelle hat den Wert $11^3 = 1331$.
- Das a an sechster Stelle hat den Wert $13^1 = 13$.
- Das b an siebter Stelle hat den Wert $17^2 = 289$.

Damit wird die GÖDEL Zahl des Wortes $w = abaccab$:

$$\begin{aligned} c(w) &= 2^1 \cdot 3^2 \cdot 5^1 \cdot 7^3 \cdot 11^3 \cdot 13^1 \cdot 17^2 \\ &= 2 \cdot 9 \cdot 5 \cdot 343 \cdot 1331 \cdot 13 \cdot 289 \\ &= 154\,367\,503\,290. \end{aligned}$$

Umgekehrt: Gegeben ist die GÖDEL Zahl $c(w) = 1\,213\,962\,750$, dann können wir das Wort w rekonstruieren durch Faktorisierung dieser Zahl:

$$\begin{aligned} 1213962750 &= 2 \cdot 9 \cdot 125 \cdot 343 \cdot 121 \cdot 13 \\ &= 2^1 \cdot 3^2 \cdot 5^3 \cdot 7^3 \cdot 11^2 \cdot 13^1 \\ &\longrightarrow abccba. \end{aligned}$$

Beispiel [1.13]

Betrachte das binäre Alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. Wir ordnen der `0` die Zahl 1 und der `1` die Zahle 2 zu. Dann haben wir:

w	$=$	`0`	$\Longrightarrow$	$c(w)$	$= 2^1 =$	2
w	$=$	`1`	$\Longrightarrow$	$c(w)$	$= 2^2 =$	4
w	$=$	`00`	$\Longrightarrow$	$c(w)$	$= 2^1 \cdot 3^1 =$	6
w	$=$	`01`	$\Longrightarrow$	$c(w)$	$= 2^1 \cdot 3^2 =$	18
w	$=$	`10`	$\Longrightarrow$	$c(w)$	$= 2^2 \cdot 3^1 =$	12
w	$=$	`11`	$\Longrightarrow$	$c(w)$	$= 2^2 \cdot 3^2 =$	36
w	$=$	`000`	$\Longrightarrow$	$c(w)$	$= 2^1 \cdot 3^1 \cdot 5^1 =$	30
w	$=$	`001`	$\Longrightarrow$	$c(w)$	$= 2^1 \cdot 3^1 \cdot 5^2 =$	150
w	$=$	`010`	$\Longrightarrow$	$c(w)$	$= 2^1 \cdot 3^2 \cdot 5^1 =$	90
w	$=$	`011`	$\Longrightarrow$	$c(w)$	$= 2^1 \cdot 3^2 \cdot 5^2 =$	450
w	$=$	`100`	$\Longrightarrow$	$c(w)$	$= 2^2 \cdot 3^1 \cdot 5^1 =$	60
w	$=$	`101`	$\Longrightarrow$	$c(w)$	$= 2^2 \cdot 3^1 \cdot 5^2 =$	300
w	$=$	`110`	$\Longrightarrow$	$c(w)$	$= 2^2 \cdot 3^2 \cdot 5^1 =$	180
w	$=$	`111`	$\Longrightarrow$	$c(w)$	$= 2^2 \cdot 3^2 \cdot 5^2 =$	900

Definition [1.7]:

Sei M die abzählbare Menge der Wörter einer (formalen) Sprache. Eine Funktion

$$g : M \longrightarrow \mathbb{N}$$

heisst **Gödelisierung**, wenn gilt:

- g ist injektiv und berechenbar.
- die Bildmenge $g(M)$ ist entscheidbar.
- die auf $g(M)$ definierte Umkehrfunktion von g ist berechenbar.

$g(m)$ heisst **Gödelnummer** von m.

Eine wichtige Aussage ist mit Hilfe der GÖDELISIERUNG leicht einzusehen.

Theorem [1.1]:

Sei Σ ein Alphabet. Da Alphabete endliche Mengen sind, ist Σ abzählbar. Dann ist jede Sprache L über Σ abzählbar.

Beweis:

Die GÖDEL Codierung ist eine Bijektion $c : L \longrightarrow \mathbb{N}$. Daher ist L abzählbar.

Eine der wichtigsten Anwendung der Gödelisierung ist die Codierung von TURING Maschinen als Zahl. Wie kommen in Abschnitt [5.4] darauf zurück.

1.2.3 Überabzählbarkeit der reellen Zahlen

Die reellen Zahlen setzen sich aus den rationalen Zahlen und allen anderen Zahlen zusammen, die sich auf der Zahlengeraden befinde, also auch Zahlen wie π, e oder $\sqrt{2}$.

Um zu sehen, dass die reellen Zahlen nicht abzählbar sind, ist es ausreichend, die reellen Zahlen in dem Intervall 0 bis 1 zu betrachten.[11] Wir führen den Beweis der Nichtabzählbarkeit durch einen Widerspruchsbeweis.

Das **zweite Diagonalargument von Georg Cantor** ist ein mathematischer Beweis dafür, dass die Menge der reellen Zahlen überabzählbar ist.

Mit seinem ersten Diagonalargument zeigte CANTOR, dass die Menge der rationalen Zahlen abzählbar ist. Wie oben gezeigt, ist es möglich, eine umkehrbar eindeutige Abbildung (eine Bijektion) zwischen der Menge der natürlichen Zahlen und der Menge der rationalen Zahlen anzugeben. Diese Abbildung erlaubt es anschaulich, alle rationalen Zahlen abzuzählen.

Durch einen Widerspruchsbeweis hat CANTOR gezeigt, dass es nicht möglich ist, eine eineindeutige Abbildung zwischen der Menge der natürlichen Zahlen $\mathbb{N}$ und der Menge der reellen Zahlen anzugeben, insbesondere ist es nicht möglich, eine *surjektive* Abbildung anzugeben. Dies hat zur Folge, dass die Menge der reellen Zahlen nicht abzählbar ist, der Terminus ist, die Menge der reellen Zahlen ist **überabzählbar Unendlich**.

Theorem [1.2]:

> Die Menge $I = [0, 1]$ der reellen Zahlen im Intervall $[0, 1]$ ist nicht abzählbar.

Beweis:

Die Menge der Zahlen im Intervall $[0, 1]$ ist sicherlich unendlich, denn diese Menge enthält die Zahlen

$$1, \frac{1}{2}, \frac{1}{4}, \ldots$$

Wir nehmen nun an, die Menge I ist abzählbar.

Dann gibt es eine bijektive Abbildung

$$f : \mathbb{N} \longrightarrow I.$$

[11] Denn wenn schon die reellen Zahlen im Intervall $[0, 1]$ nicht durchnummeriert werden können, dann ist dies natürlich auch nicht möglich für alle reellen Zahlen.

Sei

$$\begin{aligned} f(1) &= a_1, \\ f(2) &= a_2, \\ f(3) &= a_3, \\ f(4) &= a_4, \\ &\text{usw.} \end{aligned}$$

Dann können wir die Zahlen im Intervall I also in aufzählender Form schreiben:

$$I = [a_1, a_2, a_3, \ldots]\,. \tag{1.3}$$

Wir listen die Elemente $a_1, a_2, a_3 \ldots$ in einer Spalte und drücken jede Zahl durch ihren Dezimalbruch aus. Dann können wir schreiben:

$$\left.\begin{array}{rcl} a_1 & = & 0.\underline{x_{11}}x_{12}x_{13}x_{14}x_{15}\ldots \\ a_2 & = & 0.x_{21}\underline{x_{22}}x_{23}x_{24}x_{25}\ldots \\ a_3 & = & 0.x_{31}x_{32}\underline{x_{33}}x_{34}x_{35}\ldots \\ a_4 & = & 0.x_{41}x_{42}x_{43}\underline{x_{44}}x_{45}\ldots \\ a_5 & = & 0.x_{51}x_{52}x_{53}x_{54}\underline{x_{55}}\ldots \\ \vdots & \vdots & \end{array}\right\} \tag{1.4}$$

Hierbei sind alle x_{ij} Ziffern aus der Menge $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Für solche Zahlen, die zwei verschiedene Dezimalbruchentwicklungen haben, wählen wir die Entwicklung, die mit 9er endet.[12]

Wir setzen

$$b = 0.y_1y_2y_3y_4\ldots$$

Diese reelle Zahl erhalten wir aus unserer Liste (1.4) wie folgt:

$$y_i = \begin{cases} 1 & \text{falls } x_{ii} \neq 1, \\ 2 & \text{falls } x_{ii} = 1. \end{cases}$$

Die Zahl b nennt man Diagonalzahl, da sie aus den Diagonalelementen der Folgenglieder der Folge (1.4) gebildet wird. Dies geschieht so, dass sich die Zahl b in mindestens einer Stelle — nämlich gerade die Stelle x_{ii} — von jeder anderen Zahl der Liste unterscheidet. Da gemäß unserer Annahme die Menge der reellen Zahlen im Intervall $[0, 1]$ abzählbar ist, muss die Zahl b in der Liste (1.3)

[12] Es ist beispielweise $0.999999\ldots = 1.0$. Denn

$$0,333... = 3/10 + 3/100 + 3/1000 + ...$$

Diese unendliche Reihe ist konvergent und hat den Grenzwert 1/3. Deshalb ist $0.333... = 1/3$. In genau der gleichen Weise schreiben wir:

$$0,999... = 9/10 + 9/100 + 9/1000 + ...$$

Diese unendliche Reihe konvergiert ebenfalls und zwar gegen 1. Deshalb ist $0.999... = 1$.

irgendwann auftreten. Nun ist aber

$$\begin{aligned} b &\neq a_1, \text{ weil } y_1 \neq x_{11}, \\ b &\neq a_2, \text{ weil } y_2 \neq x_{22}, \\ b &\neq a_3, \text{ weil } y_3 \neq x_{33}, \\ b &\neq a_4, \text{ weil } y_4 \neq x_{44}, \\ &\vdots \qquad\qquad \vdots \end{aligned}$$

Die Zahl b ist also per Konstruktion so gebildet, dass sie von jeder Zahl in der Liste (1.4) verschieden ist. Daher taucht b in der Liste (1.4) nicht auf. Dies ist ein Widerspruch zu der Annahme, dass das Intervall I abzählbar viele Elemente hat.

Die Folge a_i aus der Liste (1.4) enthält also *nicht* alle reellen Zahlen zwischen 0 und 1.[13] Wählt man eine andere Folge, ergibt sich möglicherweise eine andere Diagonalzahl b, aber es ist gezeigt: Für jede Folge von Zahlen zwischen 0 und 1 gibt es eine Zahl zwischen 0 und 1, die nicht in dieser Folge enthalten ist. Deshalb enthält keine Folge alle reellen Zahlen zwischen 0 und 1.

Interpretiert man Folgen als Abbildungen

$$\mathbb{N} \longrightarrow [0,1],$$

dann gibt es also keine surjektive Abbildung

$$\mathbb{N} \longrightarrow [0,1].$$

Das Intervall $[0,1]$ ist deshalb weder gleichmächtig zu $\mathbb{N}$ noch endlich, daher enthält dieses Intervall überabzählbar viele Zahlen.

Wir wollen nun zeigen, dass alle Sprachen abzählbar sind. Dazu zeigen wir zunächst, dass es über einem Alphabet nur abzählbar viele Wörter gibt. Anschließend werden wir sehen, dass jede Teilmenge einer abzählbaren Menge wieder abzählbar ist.

Theorem [1.3]:

Die Menge Σ^* aller Wörter über einem Alphabet Σ ist abzählbar.

Beweis:

Sei

$$\Sigma = \{a_1, a_2, \ldots, a_n\} \tag{1.5}$$

[13] Damit ist die Surjektivität der Abbildung nicht gegeben, was eine Voraussetzung der Bijektivität darstellt.

ein Alphabet mit den $n \in \mathbb{N}$ Zeichen a_i. Gemäß Definition [1.1] ist auf jedem Alphabet eine totale Ordnung definiert. Von dieser Eigenschaft haben wir noch keinen Gebrauch gemacht. Da (1.5) also ein Alphabet ist, gibt es eine totale Ordnung auf Σ die wir mit $<<$ bezeichnen. Dann können wir ohne Einschränkung der Allgemeinheit die Zeichen in folgender Reihenfolge ordnen[14]

$$a_1 << a_2 << a_3 << \cdots << a_n$$

Man ordnet nun die Menge Σ^* der Wörter über Σ der Länge nach. Diese Ordnung beginnt daher mit dem leeren Wort λ. Dann folgen die Wörter der Länge 1, die Wörter der Länge 2 usw. Bei gleicher Länge ordnen wir die Wörter *lexikographisch*, *i.e.* zunächst bestimmt das erste Zeichen des Wortes die Reihenfolge, bei gleichem ersten Zeichen bestimmt das zweite Zeichen die Ordnung, usw. Damit ist eine Reihenfolge aller Wörter über dem Alphabet Σ festgelegt, damit ist Σ^* abzählbar.

Beispiel [1.14]

Betrachte das binäre Alphabet $\Sigma = \{0, 1\}$. Die Abzählung der Wörter kann man nach der lexikographischen Ordnung folgendermaßen durchführen:

$\mathbb{N}$		$\{0,1\}^*$	
0	$\longrightarrow$	λ	Wort der Länge 0
1	$\longrightarrow$	0	Wörter der Länge 1
2	$\longrightarrow$	1	
3	$\longrightarrow$	00	Wörter der Länge 2
4	$\longrightarrow$	01	
5	$\longrightarrow$	10	
6	$\longrightarrow$	11	
7	$\longrightarrow$	000	Wörter der Länge 3
8	$\longrightarrow$	001	
9	$\longrightarrow$	010	
10	$\longrightarrow$	011	
11	$\longrightarrow$	100	
12	$\longrightarrow$	101	
13	$\longrightarrow$	110	
14	$\longrightarrow$	111	
15	$\longrightarrow$	0000	Wörter der Länge 4

---

[14] Man beachte, dass man diesen Aspekt auch bei Alphabeten natürlicher Sprachen vorliegen hat. So ist a der erste Buchstabe, b der zweite, bis z ist der letzte Buchstabe. Auf diese Weise ordnet man *alle* Zeichen der lateinischen Buchstaben in einer Reihenfolge.

Theorem [1.4]:

Jede Teilmenge M einer abzählbaren Menge N ist abzählbar.[15]

Beweis:

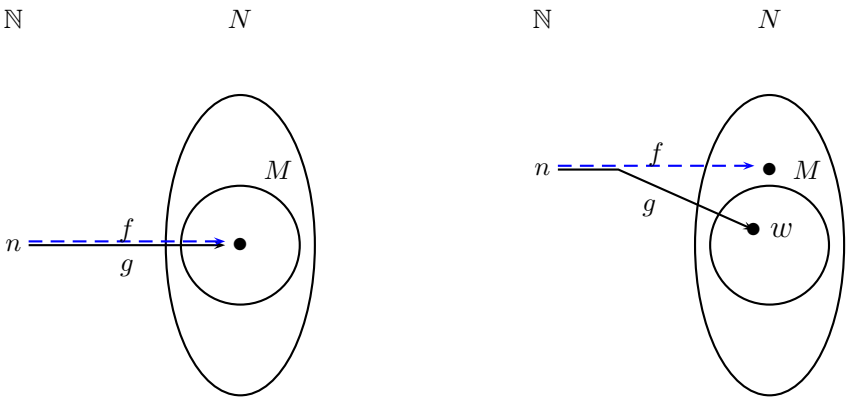
Sei die Menge N abzählbar, dann existiert eine surjektive Abbildung f mit

$$f : \mathbb{N} \longrightarrow N,$$

das heißt eine Abzählung der Elemente der Menge N.
Wir setzen

$$g(n) = \begin{cases} f(n) & \text{falls } f(n) \in M, \\ w \text{ mit } w \in M \text{ beliebig} & \text{falls } f(n) \notin N. \end{cases}$$

Da die Abbildung f surjektiv ist, kommen alle Elemente der Teilmenge $M \subset N$ als Bildmenge von f vor. Diese sind aber auch Bildelemente der Abbildung g, also ist g surjektiv.



Die obige Skizze zeigt links den Fall $f(n) \in M$ und rechts den Fall $f(n) \notin M$.

Corollar

Da für jedes Alphabet Σ die Menge aller Wörter Σ^* abzählbar ist, ist jede Sprache abzählbar.

[15] Die Tatsache, dass wir jetzt zeigen müssen, dass jede Teilmenge einer abzählbaren Teilmenge ebenfalls abzählbar ist, mag etwas trivial und selbstverständlich scheinen. De facto übertragen sich Eigenschaften von Mengen *nicht* notwendigerweise auf deren Teilmengen. Beispielweise ist die Menge der ganzen Zahlen abgeschlossen unter Subtraktion — *i.e.* die Differenz zweier ganzen Zahlen ist wieder eine ganze Zahl. Die Menge der natürlichen Zahlen ist eine echte Teilmenge der ganzen Zahlen, für diese Menge gilt *nicht* die Abgeschlossenheit unter Subtraktion.

1.2.4 Aufzählbare Sprachen

Bei der Definition der Abzählbarkeiti einer Menge — Definition [1.6] — haben wir nur die Existenz einer Durchnummerierung der Elemente dieser Menge gefordert. In der Mathematik ist es häufig der Fall, dass man auf die Existenz eines Dinges schließen kann, ohne genau zu wissen, wie dieses Ding aussieht. Dies ist oft unbefriedigend, es gibt einen Zweig in der Mathematik, diesen nennt man **Konstruktivismus**, in dem alle Teile (und Beweise) abgelehnt werden, die nicht durch **algorithmische Verfahren** abgeleitet werden.

Ein algorithmisches Verfahren — oder ein **Algorithmus** — ist eine Vorschrift,[16] gegeben durch eine endliche Menge von Regeln, die ein bestimmtes Problem löst. Abhängig von der Eingabe aus einer vorgegebenen Menge soll durch den Algorithmus schrittweise eine Ausgabe in einer bestimmten Form produziert werden. Unter einem Algorithmus können wir uns ein Computerprogramm vorstellen — *e.g.* formuliert im Code von C — denn die sogenannte **Church–Turing These** impliziert, dass alle im intuitiven Sinn algorithmischen Verfahren durch ein Computerprogramm modelliert werden können.

Da Computer nun Algorithmen verarbeiten, kann auf Rechnern prinzipiell nur der konstruktivistische Teil der Mathematik umgesetzt werden. Aus diesem Grund kann man lediglich diejenigen abzählbaren Sprachen in einem Rechner verarbeiten, deren Elemente durch konstruktives Verfahren erzeugt werden können.

Diese Überlegungen führen zur folgenden Einschränkung der Abzählbarkeit.

Definition [1.8]:

> Eine Menge M ist **aufzählbar** oder **rekursiv aufzählbar** oder **semientscheidbar**, falls es eine surjektive Abbildung
>
> $$f \;:\; \mathbb{N} \longrightarrow M$$
>
> gibt, und einen Algorithmus, der es gestattet, für jedes $n \in \mathbb{N}$ den Funktionswert $f(n)$ zu berechnen, oder falls $M = \emptyset$. Die Menge
>
> $$\{f(0), f(1), f(2), \ldots\}$$
>
> heisst **Aufzählung** von M.

[16] Das Wort *Algorithmus* ist von dem Namen des persischen Mathematikers Muhammed Musa Ibn Al Chwarismi abgeleitet, der im neunten Jahrhundert lebte. Es wird vermutet, dass er die grundlegenden Rechenmethoden zur Addition, Multiplikation, Division und Subtraktion von Dezimalzahlen entwickelt hat. Der erste überlieferte und nichttriviale Algorithmus wurde uwischen 400 und 300 v. Chr. von Euklid entwickelt. Dieses Verfahren nennt man heute den **erweiterten Euklidischen Algorithmus**. Dieser Algorithmus berechnet den größten gemeinsamen Teiler zweier natürlichen Zahlen oder das multiplikative Inverse.

Der Begriff *Algorithmus* wurde von J. F. Traub in seiner Monographie *Iterative Methodsfor the Solutions of Equations* aus dem Jahre 1964 geprägt.

Anmerkung:

Sprachen sind insbesondere auch Mengen. Daher spricht man auch von **aufzählbaren Sprachen**.

Theorem [1.5]:

Jede endliche Menge (Sprache) ist aufzählbar.

Beweis:

Sei

$$M = \{a_1, a_2, \ldots, a_m\}, \quad m \in \mathbb{N}$$

eine endliche Menge. Ein Programm zur Aufzählung dieser endlichen Menge kann nach dem folgenden Prinzip realisiert werden:

```
if(n == 1) return a_1;
if(n == 2) return a_2;
.....
if(n == k-1) return a_{k-1};
else
return a_k;
```

Hierbei ist n ein Eingabeparameter, `return` liefert das Ergebnis $f(n)$.

Theorem [1.6]:

Sei Σ ein Alphabet. Dann ist die Menge aller Wörter Σ^* über Σ aufzählbar.

Beweis:

Erzeuge zunächst das leere Wort λ. Dann die Wörter der Länge 1, dann die Wörter der Länge 2, usw. Bei gleicher Länge erzeuge die Wörter in lexikographischer Reihenfolge.

Für beliebiges $n \in \mathbb{N}$ ergibt sich $f(n)$ aus dem n–ten Schritt dieses Verfahrens.

Jede aufzählbare Sprache ist offensichtlich auch abzählbar, aber nicht umgekehrt, *i.e.* es gibt Sprachen, die nicht aufzählbar sind. Solche Sprachen sind schwierig zu beschreiben,[17] die allgemein bekannten Sprachen sind ausnahmslos aufzählbar.

[17] Quelle angeben

1.2.5 Entscheidbare Sprachen

Mit einem Aufzählungsverfahren für eine Sprache kann man für jedes Wort der Sprache nach endlich vielen Schritten nachweisen, dass das Wort ein Element der Sprache ist, denn bei der Aufzählung wird dieses Wort garantiert nach endlich vielen Schritten generiert.

Hat man jedoch ein Wort gegeben, das *nicht* ein Element einer Sprache ist, dann ist dieser Sachverhalt nicht nachweisbar. Falls die Sprache unendlich viele Wörter enthält, kann durch eine Aufzählung nicht nachgewiesen werden, dass das Wort nicht in der Sprache enthalten ist, denn dann läuft das Verfahren unendlich lang, ohne jemals alle Wörter erzeugt zu haben. Es ist nach endlicher Zeit nicht erkennbar, ob das besagte Wort vielleicht doch noch erzeugt wird.

In der Informatik sind Algorithmen relevant, die für jede Eingabe nach endlich vielen Schritten eine Antwort bezüglich eines Entscheidungskriteriums liefern. Diese Antwort kann sowohl positiv als auch negativ ausfallen. Ein typisches Problem dieser Kategorie entsteht beim Compilieren von Programmen durch einen Compiler oder Interpreter. Dieser muss zunächst entscheiden, ob die Eingabe ein zulässiges Programm ist oder nicht. Sprachen, die sich auf diese Art bearbeiten lassen, nennt man entscheidbar.

Definition [1.9]:

> Gegeben ist ein Alphabet Σ. Eine Sprache $L \subseteq \Sigma^*$ heisst **entscheidbar**, falls es einen abbrechenden Algorithmus gibt — diesen nennt man **Entscheidungsverfahren** — der für jedes $w \in \Sigma^*$ feststellt, ob $w \in L$ oder $w \notin L$.

Beispiel [1.15]

Betrachte das unäre Alphabet $\Sigma = \{x\}$. Auf diesem Alphabet betrachten wir alle Wörter mit gerader Länge, *i.e.* die Sprache

$$L = \left\{x^{2n}, n \in \mathbb{N}\right\}.$$

Die Sprache L ist entscheidbar, denn man kann ein Entscheidungsverfahren wie folgt angeben. Dieses Verfahren stellt nach endlich vielen Schritten fest, ob ein beliebiges Wort über Σ in L enthalten ist oder nicht.

> *Sei $w \in \Sigma^*$ beliebig vorgegeben. Streiche in w immer zwei x weg, solange noch ein Zeichen in dem Wort enthalten ist. Bleibt kein x übrig, dann ist $w \in L$, sonst $w \notin L$.*

Anmerkungen:

(a) Der Aspekt der Entscheidbarkeit kann zwar allgemein für eine Menge relativ zu einer Obermenge definiert werden, jedoch kann nur dann von einem Entscheidungsalgorithmus gesprochen werden, wenn diese Obermenge aufzählbar ist. Diese kann dann immer als Wertemenge interpretiert werden.

(b) Jede endliche Sprache ist entscheidbar. Dazu bildet man aus den Wörtern der Sprache eine Liste und vergleicht ein Wort w, für das entschieden werden soll, ob dieses in der Sprache enthalten ist oder nicht, mit jedem Element der Liste.

$\Longrightarrow$ Übungen [1.1], [1.10]

Theorem [1.7]:

Jede entscheidbare Sprache ist aufzählbar.

Beweis:

Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine entscheidbare Sprache über Σ. Gemäß Theorem [1.6] ist Σ^* aufzählbar. Man entscheidet mit dem nach der Voraussetzung gegebenen Entscheidungsverfahren für jedes bei der Aufzählung von Σ^* erzeugte $w \in \Sigma^*$, ob $w \in L$ oder $w \notin L$. Ignoriert man alle $w \notin L$, ergibt sich somit eine Aufzählung für L.

Aus der Aufzählbarkeit einer Sprache folgt nicht deren Entscheidbarkeit. Das Problem liegt darin, für diejenigen Wörter eine Entscheidung zu treffen, die nicht in der Sprache enthalten sind. Man kann sich durch eine Aufzählung so viele Wörter ansehen, wie man will, diese Wörter werden nicht in der Aufzählung vorhanden sein. Trotzdem kann man nie wissen, ob diese Wörter nicht doch noch später in der Aufzählung erscheinen.

Beispiel [1.16]

Wir fassen Computerprogramme und Eingaben für Computerprogramme als Wörter über dem ASCII–Alphabet auf. Dann können wir die Sprache

$$L = \{x, yx \text{ ist ein Programm, } y \text{ ist eine Eingabe, und } x \text{ stoppt bei der Eingabe von } y \text{ nach endlich vielen Schritten}\}$$

mit $L \subset ASCII^*$ zwar aufzählen, aber nicht entscheiden. Das bedeutet, es gibt keinen allgemeingültigen Algorithmus, der für ein beliebiges Programm x entscheidet, ob dieses Programm bei einer gegebenen Eingabe y stoppt oder ob es in eine unendliche Schleife gerät. Diesen Sachverhalt nennt man das **Halteproblem**.

1.2.6 Zusammenfassung

Die Menge der abzählbaren, aufzählbaren und entscheidbaren Sprachen bilden eine echte Mengenhierachie, dies ist in der Abbildung [1.3] dargestellt.

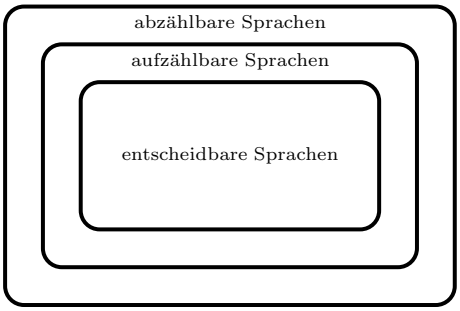


Abbildung 1.3: Mengenhierarchie der entscheidbaren, aufzählbaren und abzählbaren Sprachen.

Terminologie

In der Literatur [79] werden viele Begriffe synonym zu *berechenbar* bzw. *entscheidbar* verwendet Anstelle von entscheidbaren Problemen verwendet man die Bezeichnung **lösbare Probleme**. Verwendet werden auch zusätzliche Attribute wie *effekitv*, *rekursiv*, *algorithmisch*, die die Bedeutung dieser Begriffe aber nicht ändern.

Gleiches trifft für Resultate zu mit negativen Ergebnissen; alle Begriffe werden synonym verwendet: *unentscheidbar*, *unlösbar*, *rekursiv unentscheidbar*, *rekursiv unlösbar*, *algorithmisch unentscheidbar*, *algorithmisch unlösbar*, *effektiv unentscheidbar*, *effektiv unlösbar*. In der klassischen Mathematik verwendet man für den Begriff *Entscheidbarkeit* auch *Rekursivität*.

1.2.7 Folgerungen

Mit dem Begriff der Abzählbarkeit lassen sich eine Reihe von Folgerungen ableiten. Diese betreffen insbesondere die Frage, was Computer berechnen können und was nicht.

Da Computerprogramme endliche Folgen von Zeichen eines Alphabets (ASCII) sind, kann man die Programme als Wörter über einem Alphabet auffassen. Unter dieser Sichtweise ist die Menge der Computerprogramme eine Sprache über dem Alphabet ASCII. Wir haben gesehen, dass Sprachen im allgemeinen abzählbar sind, dies impliziert im gegenwärtigen Kontext, dass es abzählbar viele Computerprogramme gibt. Da jedes Computerprogramm ein bestimmtes Problem löst, können nur abzählbar viele Probleme durch Computer gelöst werden. Dies ist ein verschwindend geringer Anteil der Menge aller Probleme, denn es gibt überabzählbar viele Probleme.

Beispiel [1.17]

Eine einfache, überabzählbare Menge von Problemen besteht darin für jede reelle Zahl $r \in \mathbb{R}$ die folgende Funktion

$$f \ : \ \mathbb{R} \ \longrightarrow \ [0,1]$$

zu berechnen:

$$f_r(x) = \begin{cases} 1 & \text{falls } x = r \\ 0 & \text{sonst} \end{cases}$$

Die im Beispiel [1.17] präsentierte Funktion sieht etwas künstlich konstruiert aus, zeigt jedoch prinzipiell, dass es eine Vielzahl von Problemen gibt, die *nicht* durch Computerprogramme gelöst werden können.

Um eine Vorstellung über die Größenordnung des Anteils der Menge der durch Computerprogramme lösbaren Probleme an der Menge der existierenden Probleme zu erhalten, können wir den Anteil der rationalen Zahlen an den reellen Zahlen im Einheitsintervall untersuchen.

Ein alternatives Verfahren die Überabzählbarkeit der Menge der reellen Zahlen zu zeigen,[18] besteht aus folgender Überlegung. Wir betrachten wieder das Einheitsintervall $[0,1] \subset \mathbb{R}$ und nehmen an, die Menge der reellen Zahl ist abzählbar. Dann muss es möglich sein, die reellen Zahlen mit Nummern $1, 2, 3, 4, \ldots$ zu versehen, so dass man die Folge reeller Zahlen $r_1, r_2, r_3, \ldots$ in $[0,1]$ vorliegen hat. Jeder Zahl wird nun ein reelles Intervall zugewiesen, *i.e.* die Zahl r_1 liegt im Intervall I_1, die Zahl r_2 im Intervall I_2, r_3 in I_3 usw. Das erste Intervall I_1 soll

[18] Dieser alternative Beweis stammt von Richard Courant und Herbert Robbins, [55], Seite 63.

die Länge 1/10 haben, I_2 die Länge 1/100 usw. Da die Zahl r_i nicht in der Mitte des Intervalls I_i liegen muss, kann man die Intervall so wählen, dass jedes Intervall im Einheitsintervall $[0, 1]$ enthalten ist.

Da die betrachteten Zahlen $r_1, r_2, \ldots$ das Einheitsintervall vollständig ausfüllen — das ist ja unsere Annahme — müssen die gewählten Intervalle $I_1, I_2, I_3, \ldots$ das gesamte Einheitsintervall — eventuell mit Überschneidungen — überdecken, so dass die Summe ihrere Längen mindestens 1 ergibt. Nun ist aber

$$\begin{aligned} S &= \frac{1}{10} + \frac{1}{100} + \frac{1}{1000} + \cdots \\ &= \sum_{i=1}^{\infty} \frac{1}{10^i} \\ &= \frac{1}{10} \left(\sum_{i=0}^{\infty} \frac{1}{10^i} \right) \\ &= \frac{1}{10} \cdot \frac{1}{1 - \frac{1}{10}} \\ &= \frac{1}{9}. \end{aligned}$$

Im vorletzten Schritt machen wir von der geometrischen Reihe Gebrauch. Das ist nun offenbar ein Widerspruch, daher ist die Annahme falsch, die reellen Zahlen sind abzählbar.

Um nun den *Anteil* rationaler Zahlen in der Menge der reellen Zahlen des Einheitsintervalls abzuschätzen, führen wir eine sehr ähnliche Argumentation durch. Wir wissen, dass die rationalen Zahlen im Einheitsintervall $[0, 1]$ abzählbar sind. Daher können wir wie zuvor wir um jede rationale Zahl ein Intervall legen. Wir wählen jedoch die Länge des Intervalls für die i–te rationale Zahl $\frac{\epsilon}{10^i}$ mit $0 < \epsilon < 1$, ϵ klein. Als Summe der Länge aller Intervalle erhalten wir dann

$$\begin{aligned} S &= \frac{\epsilon}{10} + \frac{\epsilon}{10^2} + \frac{\epsilon}{10^3} + \cdots \\ &= \sum_{i=1}^{\infty} \frac{\epsilon}{10^i} \\ &= \sum_{i=1}^{\infty} \frac{\epsilon}{10^i} \\ &= \frac{\epsilon}{9}. \end{aligned}$$

Das bedeutet, dass die abzählbare Menge der rationalen Zahlen des Einheitsintervalls in einer Menge von Intervallen der Gesamtlänge $\epsilon/9$ eingeschlossen werden kann.

Da ϵ beliebig klein gewählt werden kann, ergibt sich als Gesamtlänge der einschließenden Intervalle der Grenzwert von $\epsilon/9$ für ϵ gegen 0, also die Gesamtlänge

0. Im Gegensatz dazu ergeben die reellen Zahlen des Einheitsintervalls die Länge 1.

Dies impliziert, dass die rationalen Zahlen einen *verschwindend* geringen Anteil an den reellen Zahlen darstellen.

Dieser Sachverhalt kann direkt auf den Anteil der durch einen Computer lösbaren Probleme an der Menge aller möglichen Probleme übertragen werden. Ihr Anteil ist daher verschwindend gering im Verhältnis zur Anzahl alle existierenden Probleme.

Beispiel [1.18]

Dass es auch praktisch relevante Probleme gibt, die nicht durch einen Computer gelöst werden können — man beachte, dass dies **prinzipiell** nicht möglich ist[19] — zeigt das folgende Beispiel, das auf das Halteproblem zurückgeführt werden kann.

Dieses Problem besteht darin, eine Testsoftware zu entwickeln, die in der Lage ist, für *jedes* Computerprogeamm dessen Korrektheit zu überprüfen. Wohlgemerkt, es geht hier nicht um die syntaktische Korrektheit. Es geht um die Korrektheit in dem Sinn, dass das Programm für jede Eingabe die richtige Ausgabe liefert. Eine solche Überprüfung bezeichnet man auch als **Programmverifikation**. Hätte man einen allgemeingültigen Programmverifikator, dann könnte jedes Programm nach der Überprüfung der syntaktischen Korrektheit durch den Compiler auf Fehlerfreiheit getestet werden. Ein typisches Teilresultat solch einer Überprüfung ist, dass das Programm keine Endlosschleifen produziert. Es hat sich demnach gezeigt, dass eine allgemeine Programmverifikation — also ein Verfahren, das für alle Programme einer Programmiersprache funktioniert — prinzipiell nicht möglich ist.

Beispiel [1.19]

Ein Beispiel eines nicht–berechenbaren Problems ist das **Domino Problem**, in der Literatur wird dies auch als *Tiling problem* bezeichnet.[20]

In diesem Problem geht es darum, eine große Fläche mit farbigen Fliesen zu kacheln. Eine Fliese ist eine quadratische Kachel, die durch die beiden Diagonalen in vier farbige Dreiecke unterteilt ist. Die Fliesen haben eine feste Orientierung und können auch nicht gedreht werden.

Eine Eingabe des Problems besteht aus der Beschreibung von endlich vielen Fliesentypen, diese Menge nennen wir T. Jeder Fliesentyp in T ist durch die

[19] Die gilt auch für Supercomputer, Quantencomputer, usw.

[20] Siehe dazu das Buch von David Harel [10], oder Martin Gardner [9], Chapter 2.

Abfolge von vier Farben bestimmet, oben links, oben rechts, unten links und unten rechts.

Abbildung 1.4: Drei Fliesentypen. [10]

Die Abbildung [1.4] zeigt drei verschiedene Fliesentypen. Das Problem besteht nun darin, ob es möglich ist, eine beliebig große (aber endliche) Fläche[21] mit den Fliesen aus T zu kacheln, und zwar so, dass die Farben an aneinanderstoßenden Kanten übereinstimmen. Von jedem Fliesentyp sind beliebig viele Exemplare vorhanden, in der Menge T gibt es aber nur eine begrenzte Anzahl von Typen.

Man stelle sich nun vor, ein Badezimmer zu fliesen. Die Eingabe T beschreibt die zur Verfügung stehenden Fliesentypen, die Farbregel von den ästhetisch–künstlerischen Ansprüchen des Innenarchitekten. Man hätte nun gerne bevor die erste Fliese an die Wand geklebt wird eine Antwort auf die Frage, ob jede Wand — von jeglicher Größe — mit den verfügbaren Fliesentypen gekachelt werden kann, ohne die Farbregeln zu verletzen.

Dieses algorithmische Problem und seine Varianten nennt man Dominoproblem oder *tiling problem*. Der Bezug zu Domino ist die an das Dominospiel angelehnte Einschränkung an aneinanderstoßende Kanten.

Die Abbildung [1.5] illustriert das Problem mit den drei Fliesentrypen aus der Abbildung [1.4] und einer 5×5–Kachelung.

Es ist leicht nachprüfbar, dass das Fliesenmuster in der Abbildung in alle Richtungen ausgedehnt werden kann, und somit liefert dies eine Kachelung für beliebig große Wände. Das gekachelte 5×5–Muster aus der Abbildung [1.5] zeigt, dass nur die drei Kacheln aus der Abbildung [1.4] verwendet werden, diese Kacheln werden nicht gedreht, und die Farbregel wird beachtet.

Wie die Abbildung [1.6] zeigt, genügt eine Abänderung der Kachel 2 und Kachel 3, dass sich die Situation drastiscb ändert.

[21] Diese ist so bemessen, dass die Länge und Breite der Fläche genau ein Vielfaches der Fliesendimension ist.

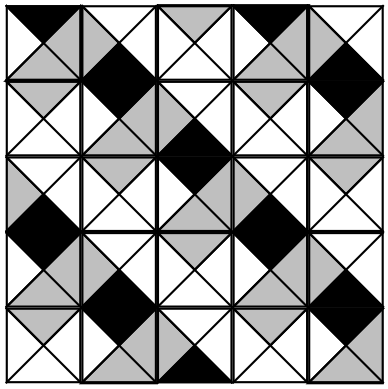


Abbildung 1.5: Die Fliesentypen aus der Abbildung [1.4] können Wände jeder Größe kacheln.

Dieses Beispiel zeigt, dass nicht einmal kleine Bereiche gekachelt werden können. Wie immer man mit dem Fliesen legen beginnt, es tritt sehr schnell die Situation ein, dass die Farben an den Kanten nicht mehr passend gelegt werden können.

Ein Algorithmus für das Dominoproblem sollte also *ja* für die drei Fliesentypen aus der Abbildung [1.4] ausgeben und *nein* für die dei Typen aus Abbildung [1.6].

In der Arbeit [[89]] aus dem Jahr 1966 wurde gezeigt, dass folgende Aussage gilt:[22]

> *Es gibt keinen Algorithmus, der das Dominoproblem löst, und es wird nie einen geben.*

Algorithmische Probleme, die keine Lösung gestatten, werden **nicht–berechenbar** genannt. Falls es sich wie im Beispiel [1.19] um ein Entscheidungsproblem handelt, nennt man solche Probleme **unentscheidbar**. Das Domino–Problem ist also unentscheidbar, *i.e.* es gibt keine Möglichkeit, einen Algorithmus für einen Computer aufzustellen, welcher zwischen Fliesentypen unterscheiden kann, mit denen alle Wände gekachelt werden können und solche, mit denen es nicht geht.

[22] Siehe auch die Arbeit von Hua Wang [126].

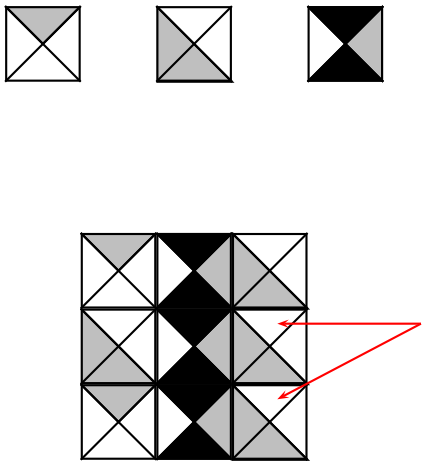


Abbildung 1.6: Fliesentypen, mit denen sich nicht einmal kleine Wände kacheln lassen.

1.3 Übungen

Übung 1.1:

Gegeben ist das binäre Alphabet $\Sigma = \{0, 1\}$. Zeigen Sie, dass die Sprache

$$L = \{w \in \{0,1\}^* \mid w = 0^n 1^n, n \in \mathbb{N}\}$$

entscheidbar ist.

Übung 1.2:

Zeigen Sie, dass die Vereinigung von abzählbar unendlich vielen abzählbar unendlichen Mengen wieder abzählbar unendlich ist.

Übung 1.3:

Überlegen Sie sich einen Algorithmus, der die Menge der natürlichen Zahlen in Binärdarstellung aufzählt.

Übung 1.4:

Zeigen Sie, dass für ein Alphabet Σ die Menge $\wp(\Sigma^*)$ überabzählbar unendlich ist.

Übung 1.5:

Sei Σ ein Alphabet. Beweisen Sie, dass nicht jede Funktion

$$f \; : \; \Sigma^* \longrightarrow \Sigma^*$$

berechenbar ist.

Übung 1.6:

Betrachte das Alphabet $\Sigma = \{a, b\}$. Betrachte die Worte $w_1 = a^2ba^3b^2$ und $w_2 = bab^2$. Bestimmen Sie:

(a) $w_1 w_2, \quad | w_1 w_2 |$

(b) $w_2 w_1, \quad | w_2 w_1 |$

(c) $w_2^2, \quad | w_1^2 |$

Übung 1.7:

Sei $\Sigma = \{a, b\}$. Beschreiben Sie verbal die folgenden Sprachen über dem Alphabet Σ.

(a) $L_1 = \{(ab)^n \mid n > 0\}$
(b) $L_2 = \{a^r ba^s ba^t \mid r, s, t \geq 0\}$
(c) $L_3 = \{a^2 b^m a^3 \mid m \in \mathbb{N}, m > 0\}$

Übung 1.8:

Seien $L_1 = \{a, ab, bb\}$ und $L_2 = \{b^2, aba\}$ zwei Sprachen über dem Alphabet $\Sigma = \{a, b\}$. Geben Sie die folgenden Sprachen an:

$$L_1L_2, L_2L_1, L_2^0, L_2^2, L_2^{-2}, (L_1L_2)^2, L_1^2L_2^2.$$

Übung 1.9:

Sei $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ das binäre Alphabet. Betrachte die Sprachen

$$L_1 = \{w \in \Sigma^* \mid w = w'\mathtt{11}, w' \in \Sigma^*\},$$
$$L_2 = \{w \in \Sigma^* \mid w = \mathtt{11}w'.w' \in \Sigma^*\}.$$

Geben Sie die folgenden Mengen an:

(a) $L_1 \cup L_2$
(b) $L_1 \cap L_2$
(c) $\overline{L_1}$
(d) $L_1 \setminus L_2$
(e) $L_2 \setminus L_1$
(f) $\overline{(L_1 \cap L_2)}$
(g) $\overline{L_1} \cap \overline{L_2}$

Übung 1.10:

Betrachte das Alphabet $\Sigma = \{a, b\}$ und die Sprache

$$L = \{w \in \Sigma^* \mid w = uu^R, u \in \Sigma^*, u^R \text{ ist das Spiegelbild von } u\}.$$

Betrachten Sie den folgenden Algorithmus. der für beliebige Worte $w \in \Sigma^*$ als Input in der folgenden Weise arbeitet:

```
while( |w| >= 2)
    {
     if(erstes Zeichen von w ungleich letztes Zeichen von w)
       entferne erstes und letztes Zeichen von w
     else
      vertausche erstes und letztes Zeichen von w
    }
if( |w| = 0)
   return nein
else
   return ja
```

(a) Ändern Sie diesen Algorithmus so ab, dass daraus ein Entscheidungsverfahren für L entsteht.

(b) Ist die Sprache L abzählbar? Wenn ja, beschreiben Sie, wie die Wörter aus L durchnummeriert werden können.

Kapitel 2

Grammatiken

Im vorigen Kapitel haben wir gesehen, dass die Menge der Schlüsselwörter einer Programmiersprache als Sprache aufgefasst werden kann. Die Menge der Schlüsselwörter ist immer endlich und relativ klein, die Programmiersprache C hat beispielweise 39 Schlüsselwörter. Eine solche Sprache wird am effektivsten dadurch verarbeit, dass einfach eine Liste aller Wörter bereitgestellt wird.

Die Menge der Bezeichner einer Programmiersprache ist ebenfalls immer endlich, da nur endlich viele Stellen für Bezeichner zulässig sind. Die Menge ist jedoch zu groß, um sie durch eine Liste bereitzustellen. In der Programmiersprache C kann ein Bezeichner maximal 31 Stellen lang sein. In jeder Stelle kann ein Zeichen stehen, 26 Groß–, 26 Kleinbuchstaben, Unterstrich oder eine von zehn Ziffern. Damit sind 53×63^{30} Bezeichner möglich. Dennoch muss ein Compiler zum Erkennen, ob ein Bezeichner zulässig ist, eine automatisch verarbeitbare Beschreibung der Sprache alle zulässigen Bezeichner zur Verfügung haben.

Wir werden verschiedene Möglichkeiten der Beschreibung unendlicher Sprachen, so dass diese in endlicher, für die automatische Verarbeitung geeigneter Form dargestellt werden können.

Zunächst werden wir die Darstellung von Sprachen mit Hilfe von **Grammatiken** betrachten. Aus natürlichen Sprachen sind Grammatiken als Regelwerke bekannt, die auf unterschiedlichen Feinheitsstufen die Struktur grammatikalisch korrekter Sätze definieren. Beispielsweise gibt es in der deutschen Sprache die Regel:

> *Ein Satz kann aus einem Subjekt bestehen, dem sich ein Prädikat und ein Objekt anschließen.*

Eine Kurzform dieser Regel, die in der Linguistik verwendet wird, lautet

> *Satz* $\longrightarrow$ *Subjekt Prädikat Objekt*

Der Satz *Der Student schreibt die Projektarbeit* hat genau diese Struktur. Hier ist *Der Student* das Subjekt, das Prädikat ist *schreibt* und *die Projektarbeit* ist

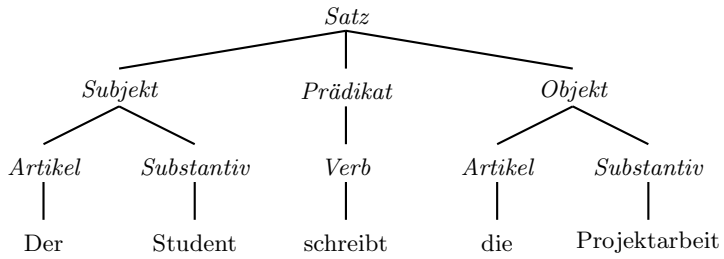


Abbildung 2.1: Grammatikalische Struktur eines Satzes der deutschen Sprache.

das Objekt. Diese Bestandteile eines Satzes können wiederum weiter strukturiert sein, beispielsweise wird die Struktur des Subjektes beschreiben durch die Regel

$$\textit{Subjekt} \longrightarrow \textit{Artikel Substantiv}$$

Weiterhin muss noch festgelegt werden, was ein Artikel ist und was ein Substantiv. Dies erreicht man durch die beiden folgenden Regeln:

$$\begin{aligned} \textit{Artikel} &\longrightarrow \text{Der} \\ \textit{Substantiv} &\longrightarrow \text{Student} \end{aligned}$$

In der Abbildung [2.1] ist eine graphische Darstellung der grammatikalischen Struktur dieses Beispielsatzes skizziert.

Es gibt natürlich weitere zulässige grammatikalische Formen für Sätze und Teilstrukturen. Diese müssen in einer vollständigen Grammatik alle durch entsprechende Regeln festgelegt sein. Die nicht mehr weiter zerlegbaren Bestandteile eines Satzes stehen üblicherweise in einer langen Liste, in der der gesamte Wortschatz einer Sprache zusammengefasst ist, im Deutschen ist dies der Duden.

Grundsätzlich ist folgendes zu beachten: Grammatiken beschreiben im Wesentlichen den syntaktischen Aufbau von Sätzen. Ob ein gemäß den grammatikalischen Regel korrekt gebildeter Satz Sinn macht oder nicht, ist eine weitere Fragestellung. Der Satz *Die Projektarbeit schreibt den Studenten* ist ein grammatikalisch korrekt gebildeter Satz, semantisch ist er jedoch problematisch. Die Untersuchung semantischer Aspekte wird hier außer Acht gelassen, da dies nicht Thema der Theorie formaler Sprachen ist.

2.1 Grundlegendes

Die Verwendung von Grammatikregeln ist nicht auf natürliche Sprachen beschränkt. Mit Grammatiken lässt sich auch der Aufbau von Compterprogrammen beschreiben. Für die Programmiersprache C gibt es beispielsweise die Regel

$$\textit{Anweisung} \longrightarrow \text{do } \textit{Anweisung} \text{ while } (\textit{Ausdruck}). \tag{2.1}$$

Definition [2.10]:

Eine **Grammatik** ist ein 4–Tupel

$$G = (N, T, P, S)\,,$$

mit

N**:** ist das Alphabet der Variablen oder Nicht–Terminalen,

T**:** ist das Alphabet der Terminalen mit $N \cap T = \emptyset$,

P**:** ist eine endliche Menge von Regeln oder Produktionen

$$u \longrightarrow v,$$

mit $u \in (N \cup T)^* \setminus T^*$ und $v \in (N \cup T)^*$. Damit muß die linke Seite der Produktionen mindestens ein Nicht–Terminalzeichen enthalten.

S**:** ist das Startsymbol, *i.e.* $S \in N$.

Anmerkung:

(a) Die terminalen Symbole werden auch **lexikalische Einheiten** genannt, die Variablen nennt man auch **grammatische Kategorien**.

(b) Man benutzt häufig die folgende abkürzende Schreibweise für die Produktionen, die die gleichen linken Seiten haben

$$u \longrightarrow v_1 \mid v_2 \mid \cdots \mid v_n \quad \text{anstatt} \quad \begin{array}{l} u \longrightarrow v_1 \\ u \longrightarrow v_2 \\ u \longrightarrow v_3 \\ \quad\vdots \\ u \longrightarrow v_k. \end{array}$$

(c) Mit den bisher betrachteten Beispielen von Grammatikregeln besteht die linke Seite stets aus einer einzelnen Variablen, *e.g.* die Regel (2.1). Im Allgemeinen ist gemäß Definition [2.10] bei einer Regel $u \longrightarrow v$ auch auf der linken Seite ein beliebiger String aus Variablen und Terminalzeichen erlaubt. Ausgeschlossen ist $u = \lambda$,

Grammatikregeln beschreiben die Struktur der Elemente einer Sprache. Ausgehend von der gröbsten Struktur, die durch die Regeln mit der Startvariablen auf der linken Seite beschrieben wird, wird die Struktur der Wörter immer feiner

zergliedert, bis man auf die nicht mehr unterteilbaren Buchstaben — das sind die Terminalzeichen – stößt. Zur Bezeichnung der Zwischenstrukturen werden Variable verwendet.

Bei dem einführenden Beispiel einer natürlichen Sprache besteht die Sprache aus den grammatikalisch korrekten Sätzen, die in diesem Kontext die Rolle der Wörter spielen. Daher wird auch anstelle des Begriffs *Wort* oft auch der Begriff *Satz* verwendet. Die Struktur der Wörter im üblichen Sinn — also Wörter wie *Student* — wird nur teilweise von Grammatiken beschrieben.[1] Die Wörter der durch eine Grammatik beschriebenen Sprache bestehen nur noch aus Terminalzeichen.

Beispiel [2.20]

Wir geben in diesem Beispiel eine Grammatik

$$G = (N, T, P, S)$$

für Bezeichner der Programmiersprache C an, die mit Buchstaben, Ziffern und Unterstrich gebildet werden können. Das erste Zeichen darf keine Ziffer sein.

Wir haben

$$\begin{aligned} N &= \{Bezeichner, BezRest, Buchstabe, Ziffer\}, \\ T &= \{a, b, c, \ldots, z, A, B, C, \ldots, Z, 0, 1, \ldots 9, \_\}, \\ S &= Bezeichner. \end{aligned}$$

Die Produktionen sind:

$$\begin{array}{rcl||rcl} Bezeichner & \longrightarrow & Buchstabe & Buchstabe & \longrightarrow & A \\ Bezeichner & \longrightarrow & \_ & Buchstabe & \longrightarrow & B \\ Bezeichner & \longrightarrow & Buchstabe\ BezRest & & \vdots & \\ Bezeichner & \longrightarrow & \_\ BezRest & Buchstabe & \longrightarrow & Z \\ BezRest & \longrightarrow & Buchstabe & Buchstabe & \longrightarrow & a \\ BezRest & \longrightarrow & \_ & Buchstabe & \longrightarrow & b \\ BezRest & \longrightarrow & Ziffer & & \vdots & \\ BezRest & \longrightarrow & Buchstabe\ BezRest & Buchstabe & \longrightarrow & z \\ BezRest & \longrightarrow & \_\ BezRest & Ziffer & \longrightarrow & 0 \\ BezRest & \longrightarrow & Ziffer\ BezRest & Ziffer & \longrightarrow & 1 \\ & & & Ziffer & \longrightarrow & 2 \\ & & & & \vdots & \\ & & & Ziffer & \longrightarrow & 9 \end{array}$$

[1] Dies geschieht durch sog. morphologische Regeln.

Die Kurzschreibweise für diese Produktionen sieht folgendermaßen aus:

$$\begin{array}{rcl} Bezeichner & \longrightarrow & Buchstabe \mid \_ \mid Buchstabe\ BezRest \mid \_\ BezRest \\ BezRest & \longrightarrow & Buchstabe\ BezRest \mid \_\ BezRest \mid Ziffer\ BezRest \\ & & \mid Buchstabe \mid \_ \mid Ziffer \\ Buchstabe & \longrightarrow & A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \\ Ziffer & \longrightarrow & 0 \mid 1 \mid 2 \mid \ldots \mid 9. \end{array}$$

2.1.1 Worterzeugung durch Grammatiken

Das Zergliedern einer vorhandenen Struktur in immer feinere Teilstrukturen geschieht durch die Anwendung der Grammatikregeln. Das Ziel dabei ist, eine grobe Struktur wie *Satz* oder *Programm* solange zu verfeinern, bis eine weitere Zergliederung nicht mehr möglich ist.

In der folgenden Definition wird beschrieben, wie eine solche induktive Verfeinerung erfolgt. Zunächst definiert man *einen* Verfeinerungsschritt: Ein beliebiges Wort, das aus Variablen und Terminalen besteht, wird mit Hilfe einer Produktionsregel der Grammatik verändert, indem ein Teilwort, welches identisch ist mit der linken Seite der Ersetzungsregel, durch die rechte Seite erstzt wird. Im zweiten Schritt wird festgelegt, wie Wörter durch eine Folge solcher Einzelschritte bearbeitet werden können.

Definition [2.11]:

Sei

$$G = (N, T, P, S)$$

eine Grammatik, $x, y \in (N \cup T)^*$.

(a) Das Wort x wird unmittelbar übergeführt in das Wort y, wenn gilt:

$$\begin{array}{rcl} x \Longrightarrow y & \Longleftrightarrow & x = x_1x_2x_3, \quad y = x_1x_2'x_3, \\ & & \text{und } x_2 \longrightarrow x_2' \in P, \text{ wobei} \\ & & x_2 \in (N \cup T)^* \setminus \{\lambda\}, x_1, x_2', x_3 \in (N \cup T)^*. \end{array}$$

(b) Das Wort x wird übergeführt in das Wort y, wenn gilt

$$\begin{array}{rcl} x \overset{*}{\Longrightarrow} y & \Longleftrightarrow & \text{Es gibt eine endliche Folge } w_0, w_1, \ldots, w_n \\ & & \text{mit } x = w_0, y = w_n \text{ und} \\ & & w_{i-1} \Longrightarrow w_i, i = 1, 2, \ldots . n \\ & & \text{Die Folge } w_0, w_1, \ldots, w_n \text{ heisst} \\ & & \text{Ableitung von } y \text{ aus } x, \text{ Schreibweise} \\ & & w_0 \Longrightarrow w_1 \Longrightarrow \cdots \Longrightarrow w_n. \end{array}$$

Bemerkungen:

1. Alternative Bezeichungen zur unmittelbaren Überführungen des Wortes x in das Wort y sind
   - x führt unmittelbar zu y
   - y wird direkt aus x abgeleitet
   - y wird in einem Schritt aus x abgeleitet.
2. Alternative Bezeichnungen für die Ableitung in *mehreren* Schritten:
   - x führt zu y
   - y ist aus x ableitbar
3. Wegen (b) sind auch Ableitungen in 0 Schritten zugelassen, *i.e.* es ist

$$x \stackrel{*}{\Longrightarrow} x.$$

Definition [2.12]:

Die von einer Grammatik $G = (N, T, P, S)$ erzeugte Sprache ist

$$L(G) = \left\{ w \in T^* \mid S \stackrel{*}{\Longrightarrow} w \right\}.$$

Eine Sprache kann durch verschiedene Grammatiken erzeugt werden, solche Grammatiken, die auf die gleiche Sprache führen, nennt man äquivalent.

Definition [2.13]:

Zwei Grammatiken G_1 und G_2 heissen äquivalent, falls

$$L(G_1) = L(G_2).$$

Beispiel [2.21]

Betrachte die Grammatik

$$G = (N, T, P, S)$$

mit $N = \{S\}, T = \{0\}$ und den beiden Produktionen

$$S \longrightarrow 0S,$$
$$S \longrightarrow 0.$$

Die Ableitungen in G haben die allgemeine Form

$$S \Longrightarrow 0S \Longrightarrow 00S \Longrightarrow \cdots \Longrightarrow 0^{n-1}S \Longrightarrow 0^n.$$

Offenbar ist die von G generierte Sprache

$$L(G) = \{0^n \mid n \geq 1\}.$$

Beispiel [2.22]

Betrachte die Grammatik

$$G = (N, T, P, S)$$

mit $N = \{S\}, T = \{0, 1\}$ und den beiden Produktionen

$$S \longrightarrow 0S1,$$
$$S \longrightarrow 01.$$

Die Ableitungen in G haben die allgemeine Form

$$S \Longrightarrow 0S1 \Longrightarrow 00S11 \Longrightarrow \cdots \Longrightarrow 0^{n-1}S1^{n-1} \Longrightarrow 0^n1^n.$$

Offenbar ist die von G generierte Sprache

$$L(G) = \{0^n1^n \mid n \geq 1\}.$$

Beispiel [2.23]

Betrachte die Grammatik $G = (N, T, P, S)$ mit

$$N = \{S, B, C\}$$
$$T = \{0, 1, 2\}$$

und den Produktionen P:

$$S \longrightarrow 0SBC \tag{2.2a}$$
$$S \longrightarrow 0BC \tag{2.2b}$$
$$CB \longrightarrow BC \tag{2.2c}$$
$$0B \longrightarrow 01 \tag{2.2d}$$
$$1B \longrightarrow 11 \tag{2.2e}$$
$$1C \longrightarrow 12 \tag{2.2f}$$
$$2C \longrightarrow 22 \tag{2.2g}$$

Die Ableitungen in dieser Grammatik haben die allgemeine Form

$$\begin{aligned} S &\overset{(2.2a)}{\Longrightarrow} 0SBC \\ &\overset{(2.2a)}{\Longrightarrow} 00SBCBC \\ &\overset{(2.2a)}{\Longrightarrow} \vdots \\ &\overset{(2.2a)}{\Longrightarrow} 0^{n-1}S(BC)^{n-1} \\ &\overset{(2.2b)}{\Longrightarrow} 0^n(BC)^n \\ &\overset{(2.2c)}{\Longrightarrow} 0^nB^nC^n \qquad (n \text{ Mal}) \end{aligned}$$

Also

$$S \overset{*}{\Longrightarrow} 0^nB^nC^n.$$

Der letzte Schritt der Ableitung zeigen wir beispielsweise für $n = 3$:

$$\begin{aligned} 000BCBCBC &\overset{(2.2c)}{\Longrightarrow} 000BBCCBC \\ &\overset{(2.2c)}{\Longrightarrow} 000BBCBCC \\ &\overset{(2.2c)}{\Longrightarrow} 000BBBCCC. \end{aligned}$$

Anschließend werden die Regeln (2.2d) und (2.2e) angewendet. Dann terminiert die Verarbeitung bei

$$S \overset{*}{\Longrightarrow} 0^n1^nC^n.$$

Für den Fall $n = 3$ sieht dies folgendermaßen aus:

$$\begin{aligned} 000BBBCCC &\overset{(2.2d)}{\Longrightarrow} 0001BBCCC \\ &\overset{(2.2e)}{\Longrightarrow} 00011BCCC \\ &\overset{(2.2e)}{\Longrightarrow} 000111CCC. \end{aligned}$$

Schließlich werden durch die Anwendung der Regeln (2.2f) und (2.2g) Terminalstrings erzeugt:

$$S \overset{*}{\Longrightarrow} 0^n1^n2^n.$$

Für den Fall $n = 3$ sieht dies folgendermaßen aus:

$$\begin{aligned} 000111CCC &\overset{(2.2f)}{\Longrightarrow} 0001112CC \\ &\overset{(2.2g)}{\Longrightarrow} 00011122C \\ &\overset{(2.2g)}{\Longrightarrow} 000111222. \end{aligned}$$

Damit ist

$$L(G) = \{0^n1^n2^n \mid n \geq 1\}.$$

Dass keine anderen Wörter als diese erzeugt werden, ist bei dieser Grammatik nicht offensichtlich. Man erkennt jedoch schnell, dass alle anderen Möglichkeiten einer Regelanwendung, die oben nicht berücksichtigt wurden, entweder in einer Sackgasse enden — *i.e.* es entstehen Wörter, die nicht nur aus Terminalzeichen bestehen — oder ebenfalls Wörter der angegebenen Sprache.

2.2 Die Chomsky–Hierarchie

Die Grammatiken werden nach dem Linguisten Noam Chomsky in vier Kategorien eingeteilt. Diese vier Klassen nennt man Typ 0, Typ 1, Typ 2 und Typ 3 Grammatiken.

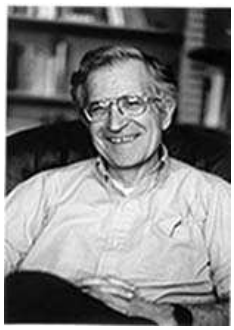


Abbildung 2.2: Noam Chomsky.

Chomsky Hierarchie:

Typ 0 Diese Grammatik unterliegt keinerlei Einschänkungen; jede Grammatik ist zunächst vom Typ 0. Hier dürfen die linke und rechte Seite der Produktionsregeln beliebige Zeichenfolgen sein, bestehend sowohl aus Variablen als auch Terminalsymbolen.

Typ 1 Typ 1 Grammatiken heißen auch **kontextsensitive Grammatiken**. Eine Typ 1 Grammatik unterliegt der Einschränkung, dass jede Regel die Form

$$w_1 A w_2 \longrightarrow w_1 B w_2$$

hat, wobei $A \in N, w_1, w_1 \in (N \cup T)^*$ und $B \in (N \cup T)^+$. Einzige Ausnahme: Wenn die Sprache das leere Wort enthält, ist die Regel $S \longrightarrow \lambda$ erlaubt, dann darf S in keiner rechten Siete der Produktionen auftreten.

Typ 2 Typ 2 Grammatiken nennt man auch **kontextfreie Grammatiken**. Sie unterscheiden sich von den Typ 0 und Typ 1 Grammatiken dadurch, dass für jede Produktionsregel

$$w_1 \longrightarrow w_2$$

die linke Seite w_1 nur aus einer einzigen Variablen bestehen darf, *i.e.* $w_1 \in N$ und $w_2 \in (N \cup T)^+$. Es gilt die gleiche Ausnahmeregel wir in Typ 1 Grammatiken.

Typ 3 Dieser Typ von Grammatiken nennt man auch **reguläre Grammatiken**. Die linke Seite der Ersetzungsregeln besteht wie bei Typ 2 aus einer einzelnen Variablen. Zusätzlich darf die rechte Seite nur eine der folgenden Formen haben

- das leere Wort
- oder ein einzelnes Terminalsymbol
- oder ein Terminalsymbol gefolgt von einer Variablen. Die Produktionen einer regulären Grammatik haben daher die Form

$$A \longrightarrow aB, \tag{2.3a}$$

oder

$$A \longrightarrow a, \tag{2.3b}$$

mit $A, B \in N, a \in T$. Die Ausnahmeregel wie in Typ 1 Grammatiken gilt hier ebenfalls.

Anmerkungen:

(a) Bei regulären und kontextfreien Grammatiken sind auch Produktionen der Form

$$A \longrightarrow \lambda$$

zulässig, wobei A nicht die Startvariable ist. Dadurch erhöht sich nicht die Anzahl der Sprachen, die man durch den entsprechenden Grammatiktyp beschreiben kann.

(b) Reguläre Grammatiken wie wir sie in der obigen Form dargestellt haben nennt man auch **rechtslineare Grammatiken**, da in den Produktionen die Variable rechts von den Terminalzeichen steht. In analoger Weise lassen sich auch **linkslineare Grammtiken** konstruieren, die ausschließlich Regeln der Form $A \longrightarrow Ba$ haben. Die rechts– und linkslinearen Grammatiken beschreiben die gleiche Klasse von Sprachen.

(c) Manche Autoren erlauben in den Definitionen für reguläre / linkslineare / rechtslineare Grammatiken überall dort, wo hier in Produktionen nur ein einzelnes Nichtterminalzeichen stehen darf, auch eine beliebige nichtleere terminale Zeichenkette. An der Erzeugungsmächtigkeit der Klassen ändert sich dadurch nichts. Dies hängt zusammen mit der sogenannten **Chomsky–Normalform**.

(d) Die Attribute *kontextfrei* und *kontextsensitiv* haben folgende Bedeutung. bei Vorliegen einer kontextfreien Regel $A \longrightarrow u$ kann die Variable A — unabhängig vom Kontext, in dem A steht — bedingungslos durch den String u ersetzt werden. Bei einer kontextsensitiven Grammatik ist es

möglich, Regeln der Form $wAv \longrightarrow wuv$ anzugeben. Dies impliziert, dass A durch u nur dann ersetzt werden kann, wenn die Variable A im Kontext zwischen w und v steht.

Die Übertragung der Eigenschaften obiger Grammatiktypen auf Sprachen liefert die folgende Definition.

Definition [2.14]:

Eine Sprache L ist vom Typ i, $i = 0, 1, 2, 3$, falls eine Grammatik von Typ i existiert, so dass $L = L(G)$.

Anmerkungen:

Der Übersichtlichkeit halber nochmals der folgende Sachverhalt:

(a) Eine Sprache L nennen wir **regulär**, genau dann, wenn sie durch eine reguläre Grammatik G erzeugt wird. Dieser Grammatiktyp hat ausschließlich Regeln P der Form

$$A \longrightarrow aB,$$
$$A \longrightarrow a,$$
$$A \longrightarrow \lambda,$$

wobei A und B Variable sind und a ein Terminalzeichen ist.

(b) Eine Sprache L nennen wir **kontextfrei**, wenn sie durch eine kontextfreie Grammatik G erzeugt wird. Eine Grammatik $G = (N, T, P, S)$ ist **kontextfrei**, falls P nur Regeln der Form

$$A \longrightarrow v, \quad A \in N, v \in (N \cup T)^+$$

enthält, *i.e.* auf der linken Seite der Produktionen darf nur eine Variable stehen, rechts darf ein beliebiger nichtleerer String von Terminalzeichen und Variablen stehen.

Der folgende Satz sagt aus, dass die CHOMSKY–Hierarchie eine echte Hierarchie von Sprachen darstellt.

Theorem [2.8]:

Gilt $0 \leq i < j \leq 3$, und ist die Grammatik G — bzw. die Sprache L — vom Typ j, so ist G (bzw. L) auch vom Typ i.

Dies impliziert, dass jede reguläre Sprache auch kontextfrei, kontextsensitiv und vom Typ 0 ist. Jede kontextfreie Sprache ist auch kontextsensitiv und vom Typ 0. Das Ziel bei der Einordnung der Grammatiken bzw. Sprachen stets den speziellen Typ zu finden.

Beispiel [2.24]

In dem Beispiel [2.20] haben wir eine Grammatik für Bezeichner einer Programmiersprache angegeben, die die folgenden Produktionen hat:

$$\begin{aligned}
\mathit{Bezeichner} &\longrightarrow \mathit{Buchstabe} \mid _ \mid \mathit{Buchstabe}\ \mathit{BezRest} \mid _\ \mathit{BezRest} \\
\mathit{BezRest} &\longrightarrow \mathit{Buchstabe}\ \mathit{BezRest} \mid _\ \mathit{BezRest} \mid \mathit{Ziffer}\ \mathit{BezRest} \\
&\quad \mid \mathit{Buchstabe} \mid _ \mid \mathit{Ziffer} \\
\mathit{Buchstabe} &\longrightarrow A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \\
\mathit{Ziffer} &\longrightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9.
\end{aligned}$$

Diese Grammatik ist kontextfrei aber nicht regulär, denn auf der linken Seite der Produktionen steht jeweils eine Variable, auf der rechten Seite stehen Strings aus Variablen, zum Beispiel *Buchstabe BezRest*. Die Frage ist, ob es eine äquivalente reguläre Grammatik gibt. Die folgenden Produktionen sind Produktionen einer Typ 3 Grammatik, daher

$$\begin{aligned}
\mathit{Bezeichner} &\longrightarrow \mathrm{A} \mid \mathrm{B} \mid \ldots \mid \mathrm{Z} \mid \mathrm{a} \mid \mathrm{b} \mid \ldots \mid \mathrm{z} \mid _ \\
&\quad \mid \mathrm{A}\ \mathit{BezRest} \mid \mathrm{B}\ \mathit{BezRest} \mid \ldots \mid \mathrm{Z}\ \mathit{BezRest} \\
&\quad \mid \mathrm{a}\ \mathit{BezRest} \mid \mathrm{b}\ \mathit{BezRest} \mid \ldots \mid \mathrm{z}\ \mathit{BezRest} \mid _\mathit{BezRest} \\
\\
\mathit{BezRest} &\longrightarrow \mathrm{A} \mid \mathrm{B} \mid \ldots \mid \mathrm{Z} \mid \mathrm{a} \mid \mathrm{b} \mid \ldots \mid \mathrm{z} \mid _ \\
&\quad \mid \mathrm{A}\ \mathit{BezRest} \mid \mathrm{B}\ \mathit{BezRest} \mid \ldots \mid \mathrm{Z}\ \mathit{BezRest} \\
&\quad \mid \mathrm{a}\ \mathit{BezRest} \mid \mathrm{b}\ \mathit{BezRest} \mid \ldots \mid \mathrm{z}\ \mathit{BezRest} \mid _\mathit{BezRest} \\
&\quad \mid \texttt{0}\ \mathit{BezRest} \mid \texttt{1}\ \mathit{BezRest} \mid \ldots \mid \texttt{9}\ \mathit{BezRest}.
\end{aligned}$$

In dieser Form haben die Produktionen der Grammatik auf der linken Seite jeweils *eine* Variable, *i.e. Bezeichner* und *BezRest*, auf der rechten Seite steht entweder ein Terminalzeichen oder ein Terminalzeichen, gefolgt von einer Variablen.

Beispiel [2.25]

Wir haben im Beispiel [2.2a] die Grammatik $G = (N, T, P, S)$ betrachtet mit

$$N = \{S\}, T = \{0\} \text{ und } P = \{S \longrightarrow 0S, S \longrightarrow 0\}.$$

Dies ist eine rechtslineare Grammatik, somit auch regulär, daher ist auch die Sprache

$$L = \{0^n \mid n \geq 1\}$$

eine reguläre Sprache.

Beispiel [2.26]

Im Beispiel [2.22] haben wir die Grammatik $G = (N, T, P, S)$ betrachtet mit

$$N = \{S\}, T = \{\mathtt{0}, \mathtt{1}\} \text{ und } P = \{S \longrightarrow \mathtt{0}S\mathtt{1}, S \longrightarrow \mathtt{01}\}.$$

Diese Grammatik ist kontextfrei, damit ist auch die Sprache

$$L = \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid w = \mathtt{0}^n\mathtt{1}^n, n \geq 1\} \tag{2.4}$$

eine kontextfreie Sprache. Wir werden sehen, dass diese Sprache nicht durch eine reguläre Grammatik erzeugt werden kann, somit ist (2.4) eine nicht–reguläre Sprache.

Beispiel [2.27]

Die Grammatik aus dem Beispiel [2.23] ist vom Typ 0. Allerdings stört nur die Regel $CB \longrightarrow BC$, ansonsten wäre die Grammatik kontextsensitiv. Diese Regel kann aber durch drei kontextsensitive Regeln ersetzt werden:

$$\begin{array}{lcl} CB & \longrightarrow & CX \\ CX & \longrightarrow & BX \\ BX & \longrightarrow & BC, \end{array}$$

wobei X eine neue Variable ist. Diese Grammatik generiert die gleiche Sprache $L = \{0^n1^n2^n \mid n \geq 1\}$ wie die ursprüngliche Grammatik, folglich können wir die Sprache L als kontextsensitive Sprache klassifizieren. Wir werden später sehen, dass L nicht durch eine kontextfreie Grammatik erzeugt werden kann.

2.2.1 Das leere Wort

In diesem Abschnitt zeigen wir Möglichkeiten auf, eine Grammatik, die das leere Wort λ nicht erzeugt, so zu ändern, dass der Typ der Grammatik gleich bleibt und dass λ zusätzlich erzeugt wird. Mit anderen Worten, der Typ einer Sprache hängt nicht davon ab, ob diese das leere Wort λ enthält oder nicht.

Typ 0: Bei Typ 0 Grammatiken müssen wir lediglich die Regel $S \longrightarrow \lambda$ hinzufügen.

Typ 1,2: Diese einfache Modifikation ist für Typ 1 und Typ 2 Grammatiken nicht anwendbar, da bei es bei Verwendung der Regel $S \longrightarrow \lambda$ vorkommen kann, dass das Startsymbol rechts auftritt. Bei kontextsensitiven und kontextfreien Grammatiken verfährt man so, dass man eine neue Startvariable S' einführt und die Regeln

$$S' \longrightarrow S \qquad \text{und} \qquad S \longrightarrow \lambda$$

der bestehenden Regelmenge hinzufügt.

Typ 3:] Für reguläre Grammatiken verfährt man folgendermaßen: Gegeben ist eine Typ Grammatik

$$G = (N, T, P, S)$$

mit $\lambda \notin L(G)$. Wir definieren eine Grammatik

$$G' = (N', T, P', S')$$

mit:

- S' ist neues Startsymbol
- $N' = N \cup \{S'\}$
- $P' = P \cup \{S' \longrightarrow w \mid S \longrightarrow w \in P\} \cup \{S' \longrightarrow \lambda\}$.

Offensichtlich bleibt der Typ der Grammatik erhalten, wenn wir die neuen Regeln hinzufügen und

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

Beispiel [2.28]

Die reguläre Grammatik

$$G = (N, T, P, S)$$

mit

$$N = \{S\}, T = \{0\}$$

und den Produktionen

$$S \longrightarrow 0S, \quad S \longrightarrow 0$$

erzeugt nicht das leere Wort.

Fügt man jetzt einfach die Regel $S \longrightarrow \lambda$ hinzu, verletzt man die Forderung, dass die Startvariable nicht mehr auf der rechten Seite der Produktionen auftreten darf. daher erhalten wir mit der obigen Konstruktion die reguläre Grammatik

$$G' = (N', V, P', S')$$

mit $N' = \{S', S\}$ und den Produktionen

$$S' \longrightarrow 0S \mid 0 \mid \lambda,$$
$$S \longrightarrow 0S \mid S$$

Dann ist

$$L(G') = \{0^n \mid n \geq 0\}.$$

$\Longrightarrow$ Übung [2.12]

2.2.2 Zusammenfassung

Die Abbildung [2.3] skizziert, wie sich die Mengen der abzählbaren, der aufzählbaren und der entscheidbaren Sprachen in die CHOMSKY–Hierarchie einfügen.

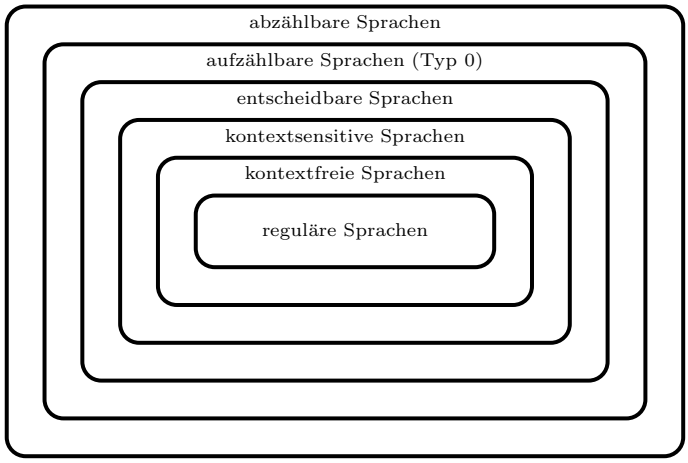


Abbildung 2.3: Die Sprachhierarchie.

Eine erste Folgerung ist, dass alle kontextsensitiven Sprachen entscheidbar sind. Somit kommen kontextsensitive Sprachen als Programmiersprachen in Frage, denn man kann damit für kontextsensitive Sprachen ein Entscheidungsverfahren entwickeln, was auf einen Compiler führt. Aus Effizienzgründen beschränkt

man sich jedoch bei der Spezifikation von Programmiersprachen auf kontextfreie Grammatiken, aus denen der sogenannte Parser abgeleitet wird. Allerdings enthalten Programme dennoch kontextsensitive Konstrukte. Beispielsweise darf eine Variable erst dann verwendet werden, wenn sie zuvor deklariert wurde. Dieser Sachverhalt wird nicht von dem Parser abgeprüft, sondern durch einen Zusatzmechanismus.

Weiterhin interessant ist, dass die Typ 0 Sprachen genau mit den aufzählbaren Sprachen übereinstimmen. Aus jeder Grammatik kann ein Aufzähliungsverfahren abgeleitet werden, in dem man alle Ableitungen aus dem Startsymbol der Länge 1, dann der Länge 2 usw. ausführt und jedesmal überprüft, ob ein Terminalwort erzeugt wurde. Diese wird dann bei der Aufzählung berücksichtigt. Aufgrund der Tatsache, dass eine Grammatik nur endlich viele Produktionsregeln enthält, gibt es zu jeder Wortlänge nur endlich viele Ableitungen mit einer bestimmten Länge.

Alle Teilmengenbeziehungen der Abbildung [2.3] sind echt. Dies lässt sich durch entsprechende Beispiele nachweisen. Beispielsweise ist die Sprache

$$L = \{0^n 1^n \mid n \geq 1\}$$

eine kontextfreie Sprache, die nicht regulär ist. Analog ist die Klasse der kontextfreien Sprachen echt in der Klasse der kontextsensitiven Sprachen enthalten, der Prototyp eine kontexsensitiven Sprache, die nicht kontextfrei ist, ist die Sprache

$$L = \{0^n 1^n 2^n \mid n \geq 1\}.$$

2.3 Weitere Formalismen zur Beschreibung kontextfreier Sprachen

Die bisher verwendete Schreibweise für Grammatiken eignet sich für die Untersuchung theoretischer Fragestellungen auf dem Papier. Für Anwendungen im Compilerbau oder bei der automatischen Verarbeitung natürlicher Sprachen werden Formalismen präferiert, die besser zur maschinellen Verarbeitung geeignet sind.

2.3.1 Backus–Naur–Form (BNF)

Die **Backus–Naur–Form** kurz BNF ist ein Formalismus zur Darstellung kontextfreier Grammatiken, der sich lediglich in der Syntax von der Theorie–orientierten Schreibweise unterscheidet. Die BNF wurde von John W. Backus und Peter Naur entwickelt, um die Programmiersprache ALGOL 60 zu entwicklen, die damit die erste durch eine Grammatik beschriebene Hochsprache war. Der Vorteil der Darstellung kontextfreier Grammatiken in Backus–Naur–Form liegt in der besseren Maschinenlesbarkeit.

Die folgenden syntaktischen Merkmale sind typische Features der BNF:

- Variable werden mit spitzen Klammern umschlossen: $< \ldots >$,
- Terminalzeichen werden ohne spezielle Kennzeichnung geschrieben
- Statt dem Pfeil in den Produktionen $\longrightarrow$ verwendet man das Symbol ::=.

Beispiel [2.29]

```
<Bezeichner>  ::= <Buchstabe> | _ |  <Bezeichner> <Buchstabe>
                 |<Bezeichner> _
                 |<Bezeichner> <Ziffer>
<Buchstabe>   ::= A | B | C | ... | Z | a | b | .... | z
<Zuffer>      ::= 0 | 1 | .... | 9
```

Diese Grammatik beschreibt Bezeichner, die mit einem Buchstaben oder Unterstrich beginnen, gefolgt von beliebig vielen Buchstaben, Unterstrichen und Ziffern,

---

Beispiel [2.30]

```
<Namensaufzaehlung>  ::=  <Name> | <Namensliste> und <Name>
<Namensliste>        ::=  <Name> | <Name> , <Namensliste>
>Name>               ::=  Hans | Maria | Otto | Thomas
```

Beschreibt Aufzählungen von Namen in der Form

Hans, Maria und Otto

2.3.2 Erweiterte Backus–Naur–Form (EBNF)

Die **Erweiterte Backus–Naur–Form** — kurz EBNF — stellt eine Erweiterung der BNF dar. Sie erweitert die BNF durch Abkürzungsmöglichkeiten für Optionalitäten und Wiederholungen. Kontextfreie Grammatiken können mit Hilfe der EBNF in einer sehr kompakten maschinenlesbaren Form dargesteltl werden.

Für die folgende Charakterisierung der EBNF sind *EBNF–Terme* die auf der rechten Seite einer EBNF Regel formulierbaren Ausdrücke. Die dabei verwendeten Grundsymbole sind Variable und Terminale, die Operatoren sind Konkatenation und Alternativbildung, aber auch die neu eingeführten Operatoren für Optionalität und die Schleifenbildung.

(a) Optionalität wird duch eckige Klammern `[....]` dargestellt. Ist beispielsweise x ein EBNF–Term, so beschreibt `[`x`]` das nullmalige oder einmalige Vorkommen eines Wortes nach dem Muster von x.

(b) Wiederholungen werden durch geschweifte Klammern $\{\ldots\}$ dargestellt. Der EBNF–Term $\{x\}$ beschreibt das nullmalige, einmalige oder mehrmalige Vorkommen von Wörtern nach dem Muster von x, beispielsweise

```
<BezRest>  ::= {<Buchstabe>|_|<Ziffer>}
```

(c) Runde Klammern werden als Strukturierungshilfen verwendet, beispielsweise

```
<Bezeichner>  ::= (<Buchstabe>|_){<Buchstabe>|_|<Ziffer>}
```

Anmerkung:

Durch die Verwendung der eckigen und geschweiften Klammern lassen sich Regeln wie

$$< X >::= [a]$$

oder

$$< X >::= \{a\}$$

formulieren. Diese Regeln erzeugen aus einer beliebigen Variablen X das leere Wort λ. Dieser Sachverhalt ändert nichts an der Tatsache, dass mit Grammatiken in EBNF–Form genau die kontextfreien Sprachen dargestellt werden können.

Vermeidung von Rekursionen

Rekursive Regeln können in EBNF oft vermieden werden indem iterative Regeln formuliert werden. Dadurch werden Sachverhalte anschaulicher und kompakter dargestellt als bei der Theorie–orientierten Schreibweise.

Beispiel [2.31]

Die folgende Produktion enthält eine Variable *Liste*, die sowohl auf der linken als auch rechten Seite der Produktion auftritt.

$$\begin{aligned} &\mathit{Liste} \longrightarrow \mathit{Element}, \mathit{Liste} \\ &\mathit{Liste} \longrightarrow \mathit{Element} \end{aligned}$$

In der EBNF Notation schreiben wir dies in der Form

```
<Liste>  ::= <Element> {,<Element>}
```

Es kann nicht jede rekursive Produktion auf diese Weise iterativ beschrieben werden, Beispielswesie kann die Regel

$$S \longrightarrow \mathtt{0}S\mathtt{1}$$

einer Grammatik für die Sprache $L = 0^n1^n$ nicht ohne Rekursion realisiert werden,

---

Mehrfachverwendung von Symbolen

In Programmiersprachen kommen die Symbole

$$\{, \}, [,], (,), <, >, |$$

selbst natürlich auch vor und haben eine völlig andere Bedeutung. Daher müssen für die Darstellung von Programmiersprachen in EBNF Konventionen festgelegt sein, die eine Verwechslung ausschließen und die Eindeutigkeit garantieren. Dieses Problem wird dadurch behoben, dass oftmals alle Terminalzeichen grundsätzlich in doppelte oder einfache Hochkommata gesetzt werden, Variable benötigen dann keine weitere Kennzeichnung mehr. Alternativ kennzeichnet

man Terminalzeichen, die in EBNF eine andere Funktionalität haben, durch einen vorangestellten Backslash.[2]

Beispiel [2.32]

Die Produktionsregel

```
<Liste> ::= \[a{,a}\]
```

beschreibt Listen der Form

$$[a], [a, a], [a, a, a], \ldots .$$

2.3.3 Syntaxdiagramme

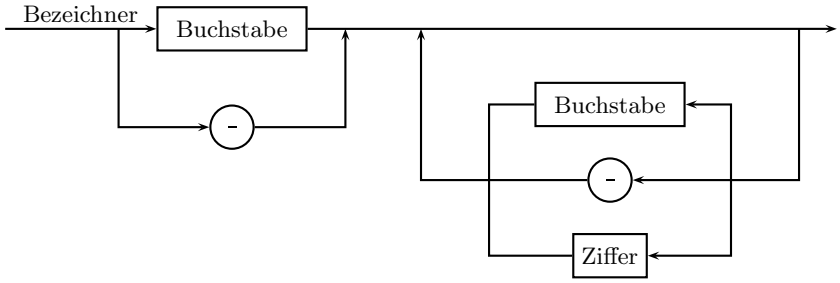
Syntaxdiagramme sind eine graphische Darstellung der erweiterten Backus–Naur–Form, die einfacher zu lesen ist. Mögliche Ableitungen der Grammatik können durch das Verfolgen von Pfaden durch die zugehörigen Syntaxdiagramme herausgefunden werden. Zur Darstellung von Programmiersprachen werden oftmas Syntaxdiagramme verwendet.

Beispiel [2.33]

Die EBNF Produktionsregel

```
<Bezeichner> ::= (<Buchstabe> | _ ) {<Buchstabe> | _ | <Ziffer> }
```

hat als Syntaxdiagramm die folgende Darstellung



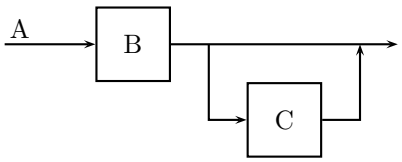
Aufbau von Syntaxdiagrammen

[2] Man nennt dieses Prozedere *Maskierung*.

- Alle Produktionen, die die gleiche Variable auf der linken Seite haben werden zusammengefaßt, indem die verschiedenen rechten Seiten als Alternativen dargestellt werden.
- Für jede Variable X wird ein gerichteter Graph erstellt mit je einer Eingangs– und Ausgangskante. Der Graph wird mit X benannt.
- Die Variablen und Terminale der rechten Seite einer Produktion werden repräsentiert durch Knoten des Graphen, man verwendet
  - Rechtecke für Variable
  - Ellipsen/Kreise für Terminale
- Konkatenationen werden durch Aneinanderreihungen, Alternativen durch Verzweigungen dargestellt.
- Die den Graphen bezeichnende Variable kann selbst als Knoten in dem Graphen auftreten.

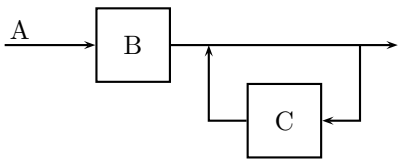
Die Darstellung der EBNF–Konstrukte Optionalität und Wiederholung in Syntaxdiagrammen wird in den folgenden Beispielen deutlich.

Optionalität



in EBNF: `<A> ::= <B>[<C>]`

Iteration



in EBNF: `<A> ::= <B>{<C>}`

Ist X die bezeichnende Variable eines Syntaxdiagramms, so entspricht jedem Weg von der Eingangs– zur Ausgangskante einem Wort, das aus X ableitbar ist. Die Rechtecke können ersetzt werden durch die Syntaxdiagramme der zugehörigen Variablen. Auf diese Weise können die aus einer Variablen ableitbaren Wörter immer genauer spezifiziert werden, solange, bis die Wörter nur noch aus

Terminalen bestehen. Beginnt man mit dem Startsymbol, so können genau die Wörter auf diese Weise erzeugt werden, die zu der von der zugehörigen Grammatik erzeugten Sprache gehören.

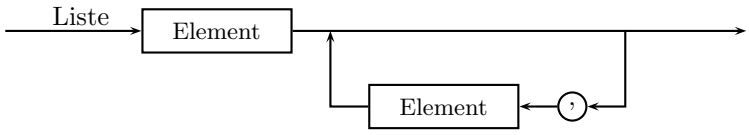
In einigen Fällen können Syntaxdiagramme in eine noch kompaktere Form umgewandelt werden. Dies demsontriert das folgende Beispiel.

Beispiel [2.34]

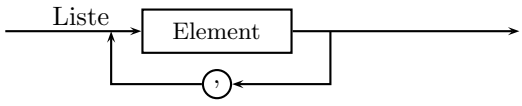
Die EBNF–Regel

```
<Liste> ::= <Element> {, <Element>}
```

kann durch das folgende Syntaxdiagramm repräsentiert werden.

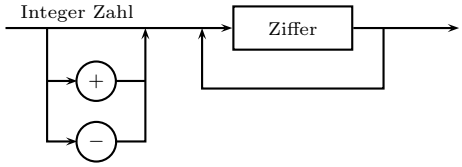


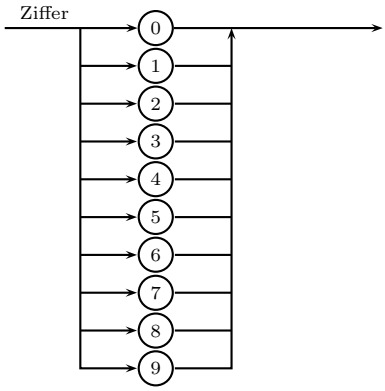
Dieses Syntaxdiagramm kann kompakter in der folgenden Form dargestellt werden:



Beispiel [2.35]

Die folgenden Syntaxdiagramme beschreiben Integerzahlen mit optionalem Vorzeichen. Führende 0en sind hierbei zugelassen.





⟹ Übung [2.14]

2.4 Übungen

Übung 2.11:

Gegeben ist die Grammatik $G = (N, T, P, S)$ mit

- $N = \{S, A, B\}$
- $T = \{0\}$
- P mit

$$\begin{array}{rcl} S & \longrightarrow & \lambda \\ S & \longrightarrow & ABA \\ AB & \longrightarrow & 00 \\ 0A & \longrightarrow & 000A \\ A & \longrightarrow & 0. \end{array}$$

- S ist Startvariable.

(a) Von welchem Typ ist diese Grammatik?

(b) Geben Sie eine Ableitung für das Wort $w = 00000$ in G an.

(c) Beschreiben Sie die Sprache $L(G)$.

(d) Geben Sie eine zu G äquivalente Grammatik G' an.

Übung 2.12:

Gegeben sind die Sprachen

$$L_1 = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid\mid w \mid_0 \mod 2 = 0 \text{ und } \mid w \mid_1 = 1\}$$
$$L_2 = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w \neq u\mathtt{00}v, w \neq u\mathtt{11}v\}$$

Geben Sie jeweils eine rechtslineare Grammatik G_i an, so dass $L_i = L(G_i)$.

Übung 2.13:

Geben Sie eine rechtslineare Grammatik G an mit $L = L(G)$, wobei

$$L = \{w \in \{\mathtt{0},\mathtt{1}\}^+ \mid \forall\, u, v \in \{\mathtt{0},\mathtt{1}\}, w \neq u\mathtt{1111}v\}.$$

Wörter in L sind also mindestens ein Zeichen lang und dürfen den Teilstring `1111` nicht enthalten. Produzieren Sie das Testwort

$$w = \mathtt{011100110001110}.$$

Übung 2.14:

Gegeben sind zwei Grammatiken mit den Regelmengen P_1 bzw. P_2, die in der erweiterten BACKUS–NAUR Form folgendermaßen aussehen:

$$P_1 : < S > ::= a[< S >]a \mid b$$
$$P_2 : < S > ::= a\{< S >\}a \mid b$$

(a) Geben Sie ein Wort an, das von der zweiten Grammatik erzeugt wird, aber nicht von der ersten Grammatik.

(b) Stellen Sie die EBNF–Regeln als Grammatikregel in theorieorientierter Form dar.

(c) Stellen Sie die EBNF–Regeln als Syntax–Diagramm dar.

Übung 2.15:

Geben Sie eine kontextfreie Grammatik an, die die Gliederung von Briefen beschreibt, die nach dem folgenden Prinzip strukturiert sind:

- Ein Brief besteht aus Einleitung, Haupttext und Schluss.
- Die Einleitung besteht aus Datum, Referenzzeichen, Empfängeranschrift und Betreff, wobei das Referenzzeichen auch fehlen kann.
- Der Haupttext besteht aus einer Anrede und einer Liste von Absätzen. Die Absätze bestehen jeseils aus einer Liste von Sätzen.
- Der Schluss besteht aus Schlussformel, Verfasser und Funktion des Verfassers.

Weiter als angegeben soll die Gliederung nicht in die Tiefe gehen. Die bei dieser Gliederung nicht unterteilbaren Bestandteile sollen als terminale Zeichen dargestellt werden. Alle angesprochenen Listen sollen nicht leer sein.

Übung 2.16:

Gegeben ist die Grammatik

$$G = (N, T, P, S)$$

mit $N = \{S\}, T = \{o, u, r, l\}$ und den Produktionen P:

$$\begin{array}{rcl} S & \longrightarrow & oSu \\ S & \longrightarrow & rSl \\ S & \longrightarrow & ou \\ S & \longrightarrow & rl \\ or & \longrightarrow & ro \\ ou & \longrightarrow & uo \\ ol & \longrightarrow & lo \\ ro & \longrightarrow & or \\ ru & \longrightarrow & ur \\ rl & \longrightarrow & lr \\ uo & \longrightarrow & ou \\ ur & \longrightarrow & ru \\ ol & \longrightarrow & lu \\ lo & \longrightarrow & ol \\ lr & \longrightarrow & rl \\ lu & \longrightarrow & ul \end{array}$$

Die Terminalen stehen für Bewegungen eines Cursors auf dem Bildschirm eines Computers, der in die vier Richtungen o – oben, u – unten, r – rechts, und l links bewegt werden kann. Es versteht sich, dass pro terminalem Zeichen eine feste Längeneinheit zurückgelegt wird.

(a) Geben Sie einen möglichst spezielle Typ für die Grammtik G an.

(b) Skizzieren Sie den Weg, der durch das Wort *oruullor* zurückgelegt wird.

(c) Geben Sie eine Ahbleitung des Wortes *oruullor* in der Grammatik an.

(d) Wie sieht die Sprache aus, die diese Grammatik erzeugt?

Übung 2.17:

Geben Sie eine kontextfreie Grammatik G an, für die gilt $L = L(G)$ und

$$L = \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid w = \mathtt{0}^i\mathtt{1}^j, i, j \geq 0, i < j\}.$$

Übung 2.18:

Gegeben sind die beiden folgenden Sprachen L_1 und L_2

$$L_1 = \{a^{2k+1}b^{2k}, k \geq 0\},$$
$$L_2 = \{(ab)c^{2k}, k > 0\}.$$

Geben Sie kontextfreie Grammatiken G_1 und G_2 an mit $L_1 = L(G_1)$ und $L_2 = L(G_2)$. Geben Sie jeweils Ableitungen zweier Beispielstrings an.

Übung 2.19:

Betrachte das Alphabet $\Sigma = \{a, b\}$ und die Sprache L über Σ, die aus allen Worten besteht, die genau ein b enthalten, *i.e.*

$$L = \{b, a^r b, ba^s, a^r ba^s; \quad r, s > 0\}.$$

Zeigen Sie, dass L eine reguläre Sprache ist.

Übung 2.20:

Betrachten Sie die folgenden Aussagen über Sprachen über einem Alphabet Σ. Argumentieren Sie, warum diese wahr oder falsch sind.

(a) Wenn die Sprachen L_1 und L_2 regulär sind, dann ist jede Sprache L mit
$$L_1 \subseteq L \subseteq L_2$$
regulär.

(b) Sei L_1 eine reguläre Sprache und L_2 eine nicht–reguläre Sprache. Dann ist $L_1 \cap L_2$ regulär.

(c) Der Durchschnitt zweier nicht–regulärer Sprachen ist stets nicht regulär.

(d) Die Menge der kontextfreien Sprachen ist gegen Komplementbildung abgeschlossen.

Hinweis:
Für ein Alphabet Σ sind die Sprachen $\emptyset$ und Σ^* durch `DFA`, `NFA` oder rechtslineare Grammatiken darstellbar und somit regulär.

Kapitel 3

Endliche Automaten, reguläre Sprachen

3.1 Einführung

Die Automatentheorie hat die Aufgabe, mit Hilfe abstrakter Automaten die unterschiedlichen Verarbeitungstechniken von mechanischen oder elektronischen Geräten zu modellieren.

Im gegenwärtigen Kontext sind Techniken interessant, mit deren Hilfe man überprüfen kann, ob ein formalsprachlicher Ausdruck syntaktische korrekt ist oder nicht. Solche Verfahren benötigt man typischerweise

- für den Kompiliervorgang eines Programms,
- bei Anwendungen wie Erkennen von Adressen durch automatische Postverteilungssysteme,
- bei der Analyse der Struktur von Webseiten.

Fasst man hierbei die Menge der zulässigen Eingaben als Wörter einer Sprache auf, so kann man die Aufgabe dadurch charakterisieren, dass ein Algorithmus — umgesetzt durch einen Automaten — alle Wörter der Sprache erkennen muss.

Wir werden sehen, dass es einen engen Zusammenhang zwischen formalen Sprachen und bestimmten Automatentypen gibt, die CHOMSKY–Hierarchie impliziert eine Hierarchie der Automatenmodelle.

Wir betrachten zunächst das Konzept der **endlichen Automaten** und sehen uns an, dass die zum Erkennen von regulären Sprachen geeignet sind. Wir zeigen dann, dass sich die Entwicklung entsprechender Erkennungsalgorithmen besonders einfach realisieren lässt, wenn man nichtdeterministische endliche Automaten betrachtet. Anschließend sehen wir uns einen alternativen Formalismus zur Beschreibung regulärer Sprachen an, die **regulären Ausdrücke**, die sich in der Praxis als Spezifikationssprache etabliert haben.

3.2 Arbeitsweise endlicher Automaten

Grammatiken beschreiben den Erzeugungsvorgang von formalen Sprachen. Bei regulären Grammatiken werden die Wörter Zeichen für Zeichen von links nach rechts generiert.

So kann man beispielsweise mit den regulären Produktionen

$$\begin{aligned} \mathit{Bezeichner} &\longrightarrow \texttt{x}\ \mathit{BezRest} \\ \mathit{BezRest} &\longrightarrow \texttt{y}\ \mathit{BezRest} \\ \mathit{BezRest} &\longrightarrow \texttt{u}\ \mathit{BezRest} \\ \mathit{BezRest} &\longrightarrow \texttt{3} \end{aligned}$$

die folgende Ableitung für den Bezeichner `xyu3` durchführen:

$$\begin{aligned} \mathit{Bezeichner} &\Longrightarrow \texttt{x}\ \mathit{BezRest} \\ &\Longrightarrow \texttt{x y}\ \mathit{BezRest} \\ &\Longrightarrow \texttt{xyu}\ \mathit{BezRest} \\ &\Longrightarrow \texttt{xyu3}. \end{aligned}$$

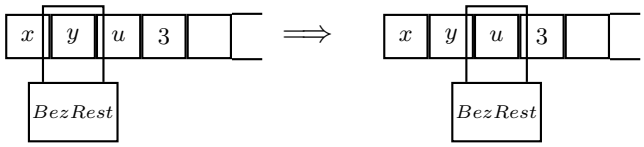
Bei jedem Ableitungsschritt fügt man durch die Anwendung einer Produktionsregel zum bisher erzeugten Präfix ein terminales Symbol hinzu sowie eine Variable (bis auf den letzten Schritt), aus der der anschließende Bezeichnerrest generiert wird.

Ein Scanner, der für ein Eingabewort überprüft, ob es ein zulässiger Bezeichner ist, ist mit einer anderen Situation konfrontiert, denn er bekommt ein fertiges Eingabewort vorgegeben. Der Scanner muss seine Aufgabe dadurch erledigen, dass er die Eingabe Zeichen für Zeichen überprüft, um am Ende seine Entscheidung treffen zu können.

Dennoch kann jeder Scannerschritt als Anwendung einer Grammatikregel aufgefasst werden. Beispielweise kann der zweite Schritt obiger Ableitung in eine Aufgabe für das Akzeptieren eines Wortes in der folgenden Weise uminterpretiert werden:

> *Ein Restwort wird als Bezeichnerrest akzeptiert, falls es mit y beginnt und anschließend ein Bezeichnerrest folgt.*

Die folgende Skizze zeigt die Situation für den Scanner vor und nach der Bearbeitung des zweiten Zeichens in dem Wort $xyu3$.



Das zu prüfende Wort $xyu3$ steht auf einem Eingabeband (Speicher), ein Lesekopf liest das Zeichen y. Die noch zu erledigende Aufgabe ist als *Zustand* der Steuereinhait des Scanners vermerkt, dieser befindet sich also in dem Zustand BezRest. Nach diesem Bearbeitungsschritt muss immer noch ein Bezeichnerrest folgen, der nun jedoch mit dem Zeichen u beginnt.

die einzelnen Arbeitsschritte eines Scanners beim zeichenweisen Lesen des Inputs können also mit Hilfe der Uminterpretation von regulären Grammatikregeln beschriebenw erden. Damit resultiert ein enger Zusammenhang zwischen Scannern und regulären Grammatiken. Wir werden sehen, dass man mit Scannern exakt die regulären Sprachen erkennen kann. Erkennen bedeutet hierbei, dass für beliebige Wörter über einem zugrundeliegenden Alphabet entschieden werden kann, ob die Wörter zu einer vorgegebenen Sprache gehören oder nicht.

Das grundlegende Prinzip eines Scanners wird in abstrakter Form durch einen **endlichen Automaten** definiert.

Ein endlicher Automat — wir verwenden oft die englische Bezeichnung *finite automaton*, insbesondere *deterministic finite automaton*, kurz DFA, oder *nondeterministic finite automaton*, kurz NFA — bekommt eine Eingabe in Form eines Wortes, das zeichenweise auf einem einseitig unendlichen Eingabeband geschrieben ist. Das Eingabeband ist unterteilt in einzelne Zellen, in jeder Zelle steht genau ein Symbol oder sie ist leer. Der Automat liest den Input mit einem Lesekopf Zeichen für Zeichen von links nach rechts bis zum Ende der Zeichenkette. Nach jedem gelesenen Zeichen kann der Automat den Zustand der Steuereinheit ändern und hat dadurch die Möglichkeit, bestimmte Informationen zu speichern. Gemäß der oben beschriebenen Interpretation kann im Zustand eine Information enthalten sein, die ausdrückt, von welcher Form das Restwort sein muss, damit es erkannt wird.

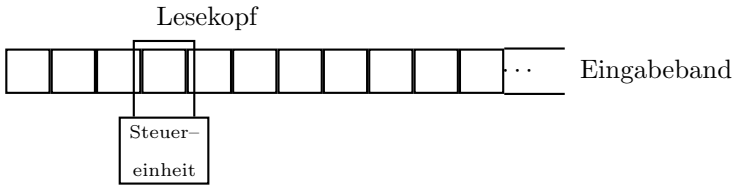


Abbildung 3.1: Schema eine endlichen Automaten.

Die Abarbeitung von Eingabewörtern wird in folgender Weise ausgeführt.

- Zu Beginn der Verarbeitung — es ist also noch kein Zeichen des Eingabestrings verarbeitet — befindet sich die Steuereinheit in einem Anfangszu-

stand.

- Der Lesekopf bewegt sich schrittweise auf dem Eingabeband nach rechts und liest die Zeichen, die in den einzelnen Zellen stehen.
- In jedem Schritt kann die Steuereinheit, abhängig von dem aktuellen Zustand und dem gelesenen Zeichen in einen anderen Zustand übergehen.
- Das Eingabewort wird von dem endlichen Automaten erkannt oder akzeptiert, wenn er sich nach dem vollständigen Lesen des Wortes in einem speziellen Acceptzustand/Endzustand befindet. Andernfalls wird das Wort nicht akzeptiert.

3.3 Grundbegriffe

Wir fassen das oben beschrieben Konzept der Arbeitsweise eines endlichen Automaten in eine formale definition. Dabei setzen wir voraus, dass die Steuereinheit nur eine endliche Anzahl von Zuständen annehmen kann. Anders formuliert, der Automat verfügt über einen sehr begrenzten Speicher, *i.e.* jeder mögliche Zustand repräsentiert eine Speichereinheit. Aus diesem Grund nennt man diese Art von Automaten auch **endliche** Automaten.

Definition [3.15]:

Ein **deterministischer endlicher Automat** — kurz DFA — ist ein 5–Tupel

$$M = (Q, \Sigma, \delta, q^s, F)$$

mit

(a) Q ist die endliche Menge der Zustände,

(b) Σ ist eine endliche Menge, das Eingabealphabet,

(c) δ ist die Übergangsfunktion

$$\delta \ : \ Q \times \Sigma \ \longrightarrow \ Q \tag{3.1a}$$

mit

$$\delta \ : \ (q, a) \longmapsto q' = \delta(q, a). \tag{3.1b}$$

(d) $q^s \in Q$ ist der Startzustand,

(e) $F \subseteq Q$ ist die Menge der Acceptzustände.

Anmerkungen:

(a) Die Übergangsfunktion (3.1b) hat die folgende Bedeutung: Wenn der Automat M sich im Zustand $q \in Q$ befindet und das Zeichen $a \in \Sigma$ liest, dann geht er in den Zustand $q' \in Q$ über und rückt den Lesekopf eine Zelle nach rechts.[1]

(b) Die Übergangsfunktion ist im allgemeinen partiell, *i.e.* es muss für ein Paar $(q, a) \in Q \times \Sigma$ nicht immer ein Funktionswert existieren. Allerdings betrachten wir in den meisten Fällen endliche Automaten mit totalem δ, dann existiert für jeden Zustand q ein Folgezustand q', wenn das Zeichen $a \in \Sigma$ verarbeitet wird. Eingabewörter werden immer komplett verarbeitet.

(c) Das Attribut *deterministisch* drückt aus, dass in jeder Situation die Arbeitsweise des Automaten eindeutig definiert ist, *i.e.* entweder gibt es genau einen Folgezuststand, oder die Maschine stoppt vorzeitig.[2] Wenn von endlichen Automaten die Rede ist, versteht man öblicherweise *deterministische endliche Automaten*, oder DFA. Im Gegensatz dazu wird bei den im folgenden Abschnitt betrachteten nichtdeterministischen endlichen Automaten (NFA) stets das Attribut *nichtdeterministisch* hinzugefügt.

Ein Scanner muss die syntaktisch korrekten Eingaben von den unkorrekten Eingaben unterscheiden können. Er muss daher erkennen können, ob eine korrekte Eingane vorliegt. Die Menge der korrekten Eingaben bildet die Sprache, die von einem Scanner erkannt wird.

Definition [3.16]:

Sei

$$M = (Q, \Sigma, \delta, q^s, F)$$

ein DFA. Das Wort $w \in \Sigma^*$ sei auf das Eingabeband geschrieben, beginnend in der ersten Zelle von links. Der Automat M startet im Anfangszustand q^s über dem ersten feld des Eingabebands.

Das Wort $w \in \Sigma^*$ wird von M erkannt oder akzeptiert, wenn M nach dem vollständigen Lesen des Eingabeworts w ein Acceptzustand annimmt.

Stoppt M vorzeitig — dies ist die Situation bei partiellen Übergangsfunktionen der Fall — dann wird das Wort nicht erkannt. Das leere Wort λ wird genau dann erkannt, wenn $q^s \in F$.

Die von M erkannt Sprache ist die Menge

$$L(M) = \{w \in \Sigma^* \mid w \text{ wird von } M \text{ erkannt}\}.$$

[1] Man beachte, dass es die Option gibt, dass $q' = q$ gilt, d.h. der Automat bleibt in diesem Fall in dem Zustand q, der sich beim Lesen des Zeichens nicht ändert.

[2] Dies ist der Fall bei partiellen Übergangsfunktionen.

Beispiel [3.36]

Wir betrachten den DFA

$$M = (Q, \Sigma, \delta, q^s, F)$$

mit der Zustandsmenge

$$Q = \{q_0, q_1, q_2\},$$

dem binären Alphabet als Eingabealphabet

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

der Startzustand ist q_0, *i.e.* $q^s = q_0$ und einen Acceptzustand $F = \{q_2\}$. Die Übergangsfunktion δ gibt man oft in Form einer Tabelle an:

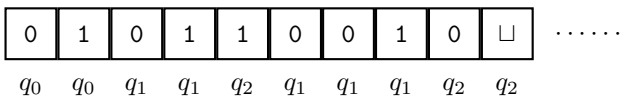
aktueller Zustand q	gelesenes Zeichen a	Folgezustand $\delta(q, a)$
q_0	0	q_0
q_0	1	q_1
q_1	0	q_1
q_1	1	q_2
q_2	0	q_2
q_2	1	q_1

Alternativ benutzen wir auch die folgende Tabellendarstellung[3]

	0	1
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_1

Wir demonstrieren die Bearbeitung zweier Wörter durch diesen DFA. Das erste Wort wird akzeptiert, das zweiet nicht. Für jede Situation (q, a) jkann aus der Tabelle der Übergangsfunktion δ der Folgezustand abgelesen werden.

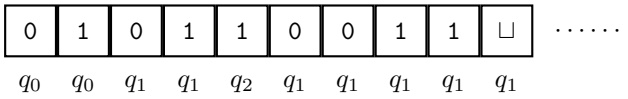
Verarbeitung des Wortes $w_1 =$ `010110010`:



[3]Diese Form der Darstellung der Übergangsfunktion wird in der Literatur überwiegend verwendet.

Der Automat befindet sich nach der Verarbeitung des Strings in dem Zustand q_2, der ein Acceptzustand ist. Daher wird der String w_1 akzeptiert.

Verarbeitung des Wortes $w_2 =$ `010110011`:



Das Wort w_1 wird akzeptiert, da die Maschine M in einem Acceptzustand terminiert. w_2 wird verworfen, da q_1 kein Acceptzustand von M ist.

Nach der Anfangsphase wechselt der Automat nur noch zwischen den Zuständen q_1 und q_2. Ein Zustandswechsel findet nur dann statt, wenn eine `1` gelesen wird, gelesene `0`en führen zu keiner Zustandsänderungen des Automaten. Die Maschine befindet sich immer in dem Zustand q_1, wenn eine ungerade Anzahl von `1`en gelesen wurde, im Zustand q_2, wenn eine gerade Anzahl gelesen wurde. Offensichtlich ist

$$L(M) = \{w \in \{0,1\}^* \mid \text{ Anzal der 1en ist gerade und } \geq 2\}$$

Der untersuchte DFA arbeitet also wie ein Paritätschecker.

3.4 Zustandsdiagramme

Die Darstellung der Übergangsfunktion δ eines DFA in Form einer Tabelle wie wir im Beispiel [3.36] verwendet haben, ist nicht sehr intuitiv, diese Form der Darstellung versagt vollständig, wenn der DFA viele Zustände hat. Fasst man die Zustände auf als knoten in einem gerichteten Graphen, dann alssen sich die Zustandsänderungen als Pfeile abbilden. Die Verarbeitung eines Inputwortes kann als Weg durch den Graphen nachvollszogen werden.

Beispiel [3.37]

Die Abbildung [3.2 zeigt das Zustandsdiagramm des Automaten aus dem Beispiel [3.36].

Das Zustandsdiagramm enthält exakt die gleichen Informationen wie die formale Definition des Automaten M aus dem Beispiel [3.36]. Jeder Zustand — das sind die Knoten des Graphen — hat für jedes Inputsymbol `0,1` genau einen auslaufenden Pfeil, dies ist der Determinismus, *i.e.* jeder Zustand des Automaten hat für jedes Inputsymbol genau einen Folgezustand.

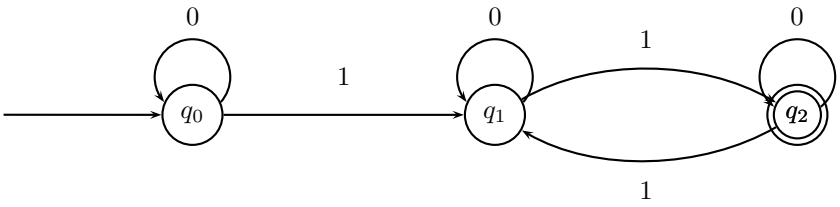


Abbildung 3.2: Das Zustandsdiagramm des endlichen Automaten M.

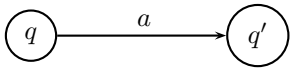
Die folgenden Regeln beschreiben den Aufbau eines Zustandsdiagramms. Das Zustandsdiagramm eines DFA

$$M = (Q, \Sigma.\delta, q^s, F)$$

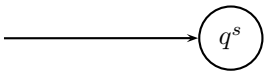
ist ein gerichteter Graph mit Beschriftung, wobei folgendes gilt:

(a) Jedem Zustand des Automaten ist genau ein Knoten zugeordnet.

(b) Für jedes Paar $(q, a) \in Q \times \Sigma$ für das die Übergangsfunktion definiert ist, führt ein Pfeil[4] mit Beschriftung a von q nach $q' = \delta(q, a)$.



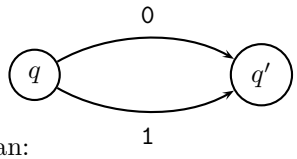
(c) Der Startzustand q^s hat einen aus dem Nichts einlaufenden Pfeil



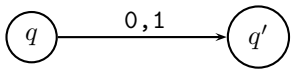
(d) Acceptzustände des Automaten werden mit einem Doppelkreis gekennzeichnet.

[4] In der Sprache der Graphen ist das eine gerichtete Kante.

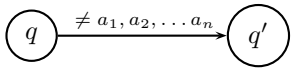
(e) Zur Vereinfachung können mehrere Übergänge zusammengefasst werden: Für



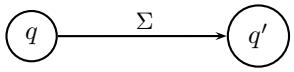
verwendet man:



Falls für alle Zeichen aus Σ außer für $a_1, a_2, \ldots a_n$ ein Pfeil von q nach q' führt, schreibt man



Falls für alle Zeichen aus Σ ein Pfeil vom Zustand q nach q' führt, schreibt man



Beispiel [3.38]

Wir betrachten den folgenden DFA:

$$M = (Q, \Sigma, \delta, q^s, F)$$

mit

$$Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{0, 1\}, q^s = q_0, F = \{q_1\}$$

und der Übergangsfunktion

	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

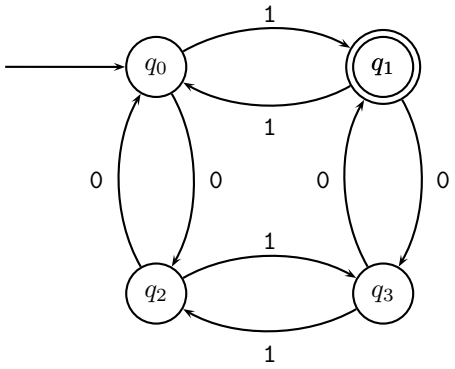


Abbildung 3.3: Zustandsdiagramm.

Bei der Betrachtung des Zustandsdiagramms wird die Arbeitsweise dieser Maschine deutlich. Dieser DFA prüft, ob due Anzahlen von `0`en bzw. `1`en in einem Bitstring gerade oder ungerade ist. Der Automat aus der Abbildung [3.3] akzeptiert alle Wörter über $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, die eine gerade Zahl von Nullen und eine ungerade Anzahl von Einsen enthalten.

Jeder der vier Zustände dieser Maschine speichert bestimmte Informationen:

q_0 Gerade Anzahl von `0`en und gerade Anzahl von `1`en gelesen.[5]

q_1 Gerade Anzahl von `0`en und ungerade Anzahl von `1`en gelesen.

q_2 Ungerade Anzahl von `0`en und gerade Anzahl von `1`en gelesen.

q_3 Ungerade Anzahl von `0`en und ungerade Anzahl von `1`en gelesen.

Durch eine geeignete Wahl der Menge der Acceptzustände F lassen sich durch diesen DFA Kombinationen von gerade und ungeraden Anzahlen abprüfen. Wählt man beispielsweise

$$F = \{q_1, q_2\},$$

dann werden alle Bitstrings erkannt, die *entweder* eine gerade Anzahl von `0`en und eine ungerade Anzahl von `1`en haben, *oder* eine ungerade Anzahl von `0`en und eine gerade Anzahl von `1` en haben.

Beispiel [3.39]

[5] Beachte, keine `0` gelesen ist gerade.

Der folgende DFA erkennt die Sprache, die aus allen Bezeichnern besteht, die mit einem Buchstaben oder einem Unterstrich beginnen, und deren Rest aus Buchstaben, Unterstrich und Ziffern zusammengesetzt ist.

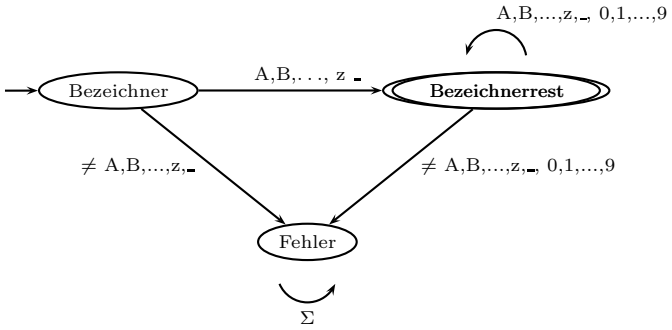


Abbildung 3.4:

Dieses Beispiel zeigt die Verwendung von Fehlerzuständen, die nicht akzeptierend sind. Die Fehlerzustände dienen dazu, bei Wörtern, die nicht erkannt werden sollen, geeignete Fehlerbehandlungen anzustoßen. Aus einem Fehlerzustand führen alle Symbole des zugrunde liegenden Alphabets Σ wieder zurück.

$\Longrightarrow$ Übung [3.30]

3.5 Nichtdeterministische endliche Automaten

Die Entwicklung der Programmiersprachen zeichnet sich dadurch aus, dass immer mehr Routinearbeiten allgemeiner Natur durch generelle Algorithmen, die bereits in der Laufzeitumgebnung oder im Compiler integriert sind, erledigt werden. Ein typisches Beispiel ist die **Rekursion**. Die Rekursion wird in modernen Hochsprachen wie Java, Pascal oder C unterstützt, nicht jedoch in älteren Sprachen wie Fortran oder Basic.

Wird ein Konzept wie die Rekursion beim Programmieren eingesetzt, so ist es zweckmäßig das Prinzip und die durch den Automatismus erledigte Verarbeitung zu verstehen, die aufwändige Detailarbeit beim Programmieren ohne Rekursion entfällt. Das Resultat sind bedeutend kürzere und besser lesbare Programme.

Ähnliche Konstrukte sind die Verwaltung der Objekthierarchien bei objektorientierten Programmiersprachen oder das eingebaute Backtracking bei logischen Programmiersprachen wie z.B. bei Prolog.

Backtracking ist ein Algorithmus, der in Situationen, in denen es mehrere Fortsetzungsmöglichkeiten beim Ablauf eines Programms gibt, sämtliche Alternativen systematisch durchgespielt werden. Dadurch wird das sogenannte nichtdeterministische Programmieren möglich, das die Lösung vieler Probleme auf hoher Abstraktionsebene mit extrem kurzem Programmcode erlaubt.

Definition [3.17]:

Ein **nichtdeterministischer endlicher Automat** — kurz NFA — ist ein 5–Tupel

$$M = (Q, \Sigma, \delta, q^s, F)$$

mit

(a) Q ist die endliche Menge der Zustände,

(b) Σ ist eine endliche Menge, das Eingabealphabet,

(c) δ ist die Übergangsrelation

$$\delta \ : \ Q \times \Sigma \ \longrightarrow \ \wp(Q) \tag{3.2a}$$

mit

$$\delta \ : \ (q, a) \longmapsto \delta(q, a) = A, \ A \subseteq Q. \tag{3.2b}$$

Hier ist $\wp(Q)$ die Potenzmenge von Q.

(d) $q^s \in Q$ ist der Startzustand,

(e) $F \subseteq Q$ ist die Menge der Acceptzustände.

Ein Wort $w \in \Sigma^*$ wird von M akzeptiert und ist ein Element der von M akzeptierten Sprache $L(M)$, wenn M, beginnend im Anfangszustand mit dem Lesekopf auf dem ersten Zeichen von w mindestens eine Möglichkeit hat, das Wort w vollständig zu verarbeiten und anschließend in einem Acceptzustand zu terminieren.

Anmerkungen:

(a) Die Definition der Übergangsrelation eines NFA, die Abbildung (3.2a), impliziert, dass es für ein Paar $(q, a) \in Q \times \Sigma$ keinen, einen oder auch mehrere Folgezustände geben kann, denn die Potenzmenge $\wp(Q)$ enthält alle Teilmengen von Q, dazu gehört die leere Menge $\emptyset$, die einelementigen und die mehrelementigen Teilmengen.

(b) Für die Akzeptanz eines Wortes genügt es, dass ein Verarbeitungszweig des NFA nach Abarbeitung des Wortes in einem Acceptzustand terminiert.

(c) Offensichtlich ist jeder deterministische endliche Automat auch ein nichtdeterministischer Automat, jedoch nicht umgekehrt.

(d) Nichtdeterministische endliche Automaten lassen sich wie die DFA durch Zustandsdiagramme beschreiben. Die Unterschiede liegen darin, dass aus einem Zustand mehrere Pfeile mit der gleichen Beschriftung ausgehen

können. Der andere Unterschied zu einem Zustandsdiagramm eines DFA ist, dass ein Zustand q keinen ausgehend Pfeil für ein Zeichen $a \in \Sigma$ haben kann, als keinen Folgezustand.

Beispiel [3.40]

Betrachte das binäre Alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ und die Sprache

$$L = \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid w = w'\mathtt{00} \text{ oder } w = w'\mathtt{01}, w' \in \Sigma^*\}, \tag{3.3}$$

i.e. die Menge aller Bitstrings, in denen an vorletzter Stelle eine `0` steht.

Der NFA, der diese Sprache akzeptiert ist formal

$$N = (Q, \Sigma, \delta, q^s, F)$$

mit den drei Zuständen

$$Q = \{q_0, q_1, q_2\},$$

sowie $\Sigma = \{\mathtt{0}, \mathtt{1}\}, q^s = q_0, F = \{q_2\}$ und der Übergangsrelation δ:

	`0`	`1`
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

Das Zustandsdiagramm dieses nichtdeterministischen endlichen Automaten ist in der Abbildung [3.5] dargestellt.

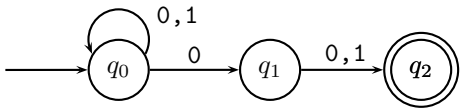


Abbildung 3.5: Das Zustandsdiagramm des NFA, der die Sprache (3.3) akzeptiert.

Die Art und Weise, wie der NFA aus der Abbildung [3.5] die Eingabestrings verarbeitet, kann man sich anhand von Beispielstrings klarmachen.

- Betrachte den String $w_1 = \mathtt{100101} \in \{\mathtt{0}, \mathtt{1}\}^*$. Die Verarbeitung dieses Inputstrings kann man sich durch einen Ableitungsbaum klarmachen, der in der Abbildung [3.6] dargestellt ist.

  Die Verarbeitung des Strings $w_1 = \mathtt{100101}$ durch den NFA kann man so lesen, dass jedesmal, wenn eine `0` gelesen wird, ein neuer Verarbeitungsstring aufgebaut wird, der im Zustand q_2 endet.[6] Wenn die gelesene `0` das

[6] Der Automat 'vermutet', dass dies die `0` der vorletzten Stelle ist.

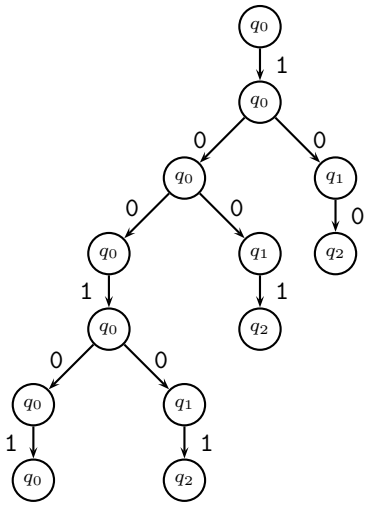


Abbildung 3.6: Ableitungsbaum des Strings $w_1 = \texttt{100101}$ durch den NFA.

vorletzte Zeichen ist, terminiert die Verarbeitung in dem Acceptzustand q_2 und der String wird akzeptiert. Folgen weitere Zeichen und ist ein Verarbeitungszweig in dem q_2–Zustand, so stirbt dieser Zweig ab und wird nicht weiter betrachtet.

- Betrachte den String $w_2 = \texttt{001110}$. Der Verabeitungsbaum dieses Strings ist in der Abbildung [3.7] dargestellt.

Für die Sprache (3.3) lässt sich auch ein deterministischer endlicher Automat angeben. Das Zustandsdiagramm dieses DFA ist in der Abbildung [3.8] dargestellt.

$\Longrightarrow$ Übung [3.31]

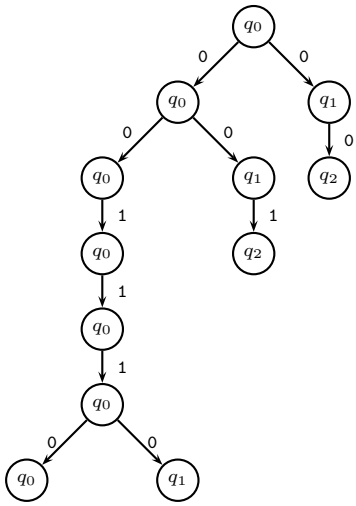


Abbildung 3.7: Verarbeitungsbaum für den String $w_2 = \texttt{001110}$ durch den NFA.

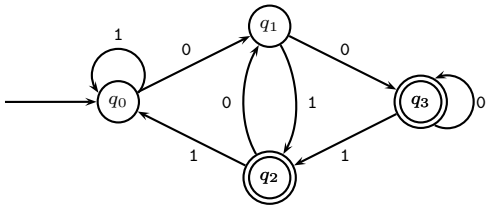


Abbildung 3.8: Ein DFA, der die Sprache (3.3) akzeptiert.

3.6 Reguläre Sprachen und endliche Automaten

Sowohl Grammatiken als auch Automaten (Akzeptoren) beschreiben Sprachen, dies jedoch unter jeweils anderen Blickwinkeln.

In dem folgenden Satz zeigen wir, dass DFA genau die regulären Sprachen akzeptieren, *i.e.* die Sprachen, die von regulären Grammatiken erzeugt werden. Weiterhin sagt der Satz aus, dass DFA und NFA äquivalent sind, genauer, jeder NFA kann durch einen DFA simuliert werden. Dieses Ergebnis ist überraschend, denn dies impliziert, dass NFA nicht mächtiger sind als DFA, beide Konzepte sind gleichmächtig.

Theorem [3.9]:

Sei $L \subseteq \Sigma^*$ eine Sprache. Die folgenden Aussagen sind äquivalent:

1. L ist regulär,
2. Es gibt einen nichtdeterministischen endlichen Automaten, der L erkennt.
3. Es gibt einen deterministischen endlichen Automaten, der L erkennt.

Beweis:

Wir führen den Beweis des Satzes [3.9] durch einen Ringschluss, *i.e.* wir zeigen[7]

$$(1) \Longrightarrow (2) \Longrightarrow (3) \Longrightarrow (1).$$

$(1) \Longrightarrow (2)$.

Wir müssen zeigen: Gegeben ist eine reguläre Sprache, *i.e.* gegeben ist eine reguläre Grammatik

$$G = (N, T, P, S)$$

mit $L = L(G)$. Dann gibt es einen nichtdeterministischen endlichen Automaten N, mit $L = L(N)$. Wir geben einen konstruktiven Beweis.

Sei also $G = (N, T, P, S)$ eine reguläre Grammatik mit $L = L(G)$. Die Produktionen einer regulären Grammatik haben gemäß den Gln. (2.3a) und (2.3b) die Form

$$A \longrightarrow aB, \qquad \text{oder} \qquad A \longrightarrow a,$$

wobei $A, B \in N$ und $a \in \Sigma$.

Wir ändern die Grammatik G wie folgt zu einer Grammatik G', die äquivalent zu einer regulären Grammatik ist, also die erzeugt Sprache nicht ändert.

- Wir wählen eine neue Variable E.
- Füge eine neue Produktion $E \longrightarrow \lambda$ zu P hinzu.
- Ersetze in P alle Regeln der Form $A \longrightarrow \mathtt{a}$ durch $A \longrightarrow \mathtt{a}E$.

Damit erhalten wir eine Grammatik $G' = (N', T, P', S)$ und es gilt $L(G) = L(G')$.

Der folgende NFA $N = (Q, \Sigma, \delta, q^s, F)$ liefert den gewünschten Automaten.

[7] Eigentlich zeigt man die Äquivalenz zweier Aussagen A und B, indem man die beiden Implikationen $A \Longrightarrow B$ und $B \Longrightarrow A$ zeigt. Aufgrund der Transitivität der Implikation — das bedeutet, wenn $A \Longrightarrow B$ und $B \Longrightarrow C$, dann folgt auch $A \Longrightarrow C$ — folgen die restlichen Implikationen aus dem Ringschluss. Das sind die Implikationen $(1) \Longrightarrow (3), (2) \Longrightarrow (1)$ und $(3) \Longrightarrow (2)$.

1. Wir definieren die Menge der Zustände Q durch

$$Q = \{q_A, A \in N\} \cup \{q_E\}, \qquad E \notin N,$$

i.e. wir ordnen jeder Variablen der regulären Grammatik einen Zustand des Automaten zu, zu diesem Zweck setzen wir

$$A \longrightarrow q_A, \quad \forall\, A \in N.$$

Wie oben erwähnt fügen eine weitere Variable E zur ursprünglichen Grammatik hinzu, welche auf den Zustand q_E führt. Durch diese Variable lassen sich die Acceptzustände des NFA modellieren.

2. $\Sigma = T$.

3. Der Startzustand des Automaten ist der Zustand q_S, wobei S die Startvariable der Grammatik ist.

4. Die Menge der Acceptzustände ist

$$F = \begin{cases} \{q_S, q_E\} & \text{falls } S \longrightarrow \lambda \in P, \\ \{q_E\} & \text{sonst .} \end{cases}$$

5. Die Übergangsrelation δ definieren wir durch

$$\delta(q_A, \mathtt{a}) = \{q_B \ \text{ mit } A \longrightarrow \mathtt{a}B \in P\}\,.$$

Da die Grammatik G im allgemeinen mehrere Übergänge der Form $A \longrightarrow \mathtt{a}B_1$ und $A \longrightarrow \mathtt{a}B_2$ haben kann, ist dies die Übergangsrelation eines nichtdeterministischen endlichen Automaten.

Anders ausgedrückt: Bei der Ableitung eines Wortes in der Grammatik kann die Regel $A \longrightarrow \mathtt{a}B_1$ oder die Regel $A \longrightarrow \mathtt{a}B_2$ für die Ersetzung der Variablen A angewendet werden. Der Automat 'rät' durch den Nichtdeterminisums, welche Regel angewendet wird.

Man zeigt durch Induktion über die Wortlänge von w, für alle $w \in T^*$ und für alle $A \in N$ gilt, dass

$$q_A \in \delta(q_s, w) \iff S \overset{|w|}{\underset{G}{\Longrightarrow}} wY.$$

Für $w \neq \lambda$ folgt dann:

$$\begin{aligned} w \in L &\iff S \overset{|w|}{\underset{G}{\Longrightarrow}} wZ \\ &\iff q_Z \in \delta(q_s, w) \\ &\iff \delta(q_s, w) \in F \neq \emptyset \\ &\iff w \in L(D). \end{aligned}$$

Beispiel [3.41]

Wir betrachten die reguläre Grammatik

$$G = (N, T, P, S)$$

mit der Menge der Variablen $N = \{S, X\}$, dem Terminalalphabet $T = \{\mathtt{0}, \mathtt{1}\}$, der Startvariablen S und den Produktionen P:

$$\begin{aligned} S &\longrightarrow \mathtt{0}S, \\ S &\longrightarrow \mathtt{1}S, \\ S &\longrightarrow \mathtt{0}X, \\ X &\longrightarrow \mathtt{0}, \\ X &\longrightarrow \mathtt{1}. \end{aligned}$$

Wir fügen eine neue Variable E hinzu sowie drei neue Produktionen

$$X \longrightarrow \mathtt{0}E, \quad X \longrightarrow \mathtt{1}E, \qquad E \longrightarrow \lambda.$$

Die beiden Regeln $X \longrightarrow \mathtt{0}$ und $X \longrightarrow \mathtt{1}$ werden entfernt. Damit haben wir die modifizierte Grammatik:

$$G' = (N', T, P', S)$$

mit

$$\begin{aligned} N' &= \{S, X, E\} \\ T &= \{\mathtt{0}, \mathtt{1}\}, \end{aligned}$$

und den Produktionen P' mit

$$\begin{aligned} S &\longrightarrow \mathtt{0}S, \\ S &\longrightarrow \mathtt{1}S, \\ S &\longrightarrow \mathtt{0}X, \\ X &\longrightarrow \mathtt{0}E, \\ X &\longrightarrow \mathtt{1}E, \\ E &\longrightarrow \lambda. \end{aligned}$$

Wir konstruieren einen NFA N, der die gleiche Sprache akzeptiert.

$$N = (Q, \Sigma, \delta, q^s, F)$$

mit

- der Menge der Zustände

$$Q = \{q_S, q_X, q_E\},$$

- dem Inputalphabet
$$\Sigma = \{0, 1\},$$
- dem Startzustand
$$q^s = q_S,$$
- der Menge der Acceptzustände
$$F = \{q_E\}$$
- und der Übergangsrelation δ:
  - da $S \longrightarrow 0S \in P'$, ist $\delta(q_S, 0) = q_S$
  - da $S \longrightarrow 1S \in P'$, ist $\delta(q_S, 1) = q_S$
  - da $S \longrightarrow 0X \in P'$, ist $\delta(q_S, 0) = q_X$
  - da $X \longrightarrow 0E \in P'$, ist $\delta(q_X, 0) = q_E$
  - da $X \longrightarrow 1E \in P'$, ist $\delta(q_X, 1) = q_E$

  In Form einer Tabelle lautet die Übergangsrelation:

	0	1
q_S	$\{q_S, q_X\}$	$\{q_S\}$
q_X	$\{q_E\}$	$\{q_E\}$
q_E	$\emptyset$	$\emptyset$

  Dies ist der NFA aus dem Beispiel [3.40] mit dem Zustandsdiagramm

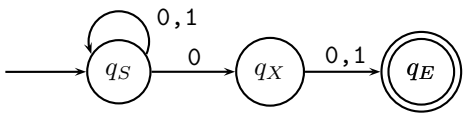


Abbildung 3.9: Das Zustandsdiagramm des NFA.

---

(2) $\Longrightarrow$ (3).

Konstruktion eines DFA aus einem NFA

Zu zeigen ist, gegeben ist ein NFA N, wir konstruieren einen DFA M, der die Arbeitsweise von N simuliert.

Sei also

$$N = (Q, \Sigma, \delta, q^s, F)$$

gegeben. Der folgende DFA ist äquivalent zu N:

$$M = (Q', \Sigma, \delta', q^{s'}, F')$$

mit

- der Menge der Zustände $Q' = \wp(Q)$,
- dem Inputalphabet Σ,
- dem Startzustand $q^{s'} = \{q^s\}$,
- der Menge der Acceptszustände:

$$F' = \{A \in Q \mid \text{ es gibt ein } q \in A \text{ mit } q \in F\}.$$

- der Übergangsfunktion

$$\delta'(A, a) = \{p \in Q \mid \text{ es gibt ein } q \in A \text{ mit } \delta(p, a) = q\}.$$

Beispiel [3.42]

Wir wandeln den NFA aus dem Beispiel [3.41] um in einen äquivalenten DFA, dabi verwenden wir das obige Konstruktionsverfahren.

Gegeben ist der NFA

$$N = (Q, \Sigma, \delta, q^s, F)$$

mit

- der Menge der Zustände

$$Q = \{q_S, q_X, q_E\},$$

- dem Inputalphabet

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

- dem Startzustand

$$q^s = q_S,$$

- der Menge der Acceptzustände

$$F = \{q_E\}$$

- und der Übergangsrelation δ:

	0	1
q_S	$\{q_S, q_X\}$	$\{q_S\}$
q_X	$\{q_E\}$	$\{q_E\}$
q_E	$\emptyset$	$\emptyset$

Wir konstruieren nun den DFA mit dem obigen Konstruktionsverfahren, der diesen NFA simuliert. Also:

$$M = (Q', \Sigma, \delta', q^{s'}, F')$$

mit

$$Q' = \wp(Q) = \{\emptyset, \{q_S\}, \{q_X\}, \{q_E\}, \{q_S, q_X\}, \{q_S, q_E\}, \{q_X, q_E\}, \{q_S, q_X, q_E\}\}\,.$$

Der Startzustand ist $q^{s'} = \{q_S\}$ und die Menge der Acceptzustände ist:

$$F' = \{\{q_E\}, \{q_S, q_E\}, \{q_X, q_E\}, \{q_S, q_X, q_E\}\}.$$

Die Übergangsfunktion erhalten wir zu:

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_S\}$	$\{q_S, q_X\}$	$\{q_S\}$
$\{q_X\}$	$\{q_E\}$	$\{q_E\}$
$\{q_E\}$	$\emptyset$	$\emptyset$
$\{q_S, q_X\}$	$\{q_S, q_X, q_E\}$	$\{q_S, q_E\}$
$\{q_S, q_E\}$	$\{q_S, q_X\}$	$\{q_S\}$
$\{q_X, q_E\}$	$\{q_E\}$	$\{q_E\}$
$\{q_S, q_X, q_E\}$	$\{q_S, q_X, q_E\}$	$\{q_S, q_E\}$

Das Zustandsdiagramm dieses DFA ist in der Abbildung [3.9] dargestellt. Der Einfachheit halber nennen wir die Zustände des DFA um, wir setzen

$$\begin{aligned}
\emptyset &= q_0,\\
\{q_S\} &= q_1,\\
\{q_X\} &= q_2,\\
\{q_E\} &= q_3,\\
\{q_S, q_X\} &= q_4,\\
\{q_S, q_E\} &= q_5,\\
\{q_X, q_E\} &= q_6,\\
\{q_S, q_X, q_E\} &= q_7.
\end{aligned}$$

Die Zustände q_3, q_5, q_6 und q_7 sind Acceptzustände. Damit können wir die oben aufgeführte Übergangsfunktion schreiben in der Form

	0	1
q_0	q_0	q_0
q_1	q_4	q_1
q_2	q_3	q_3
q_3	q_0	q_0
q_4	q_7	q_5
q_5	q_4	q_1
q_6	q_3	q_3
q_7	q_7	q_5

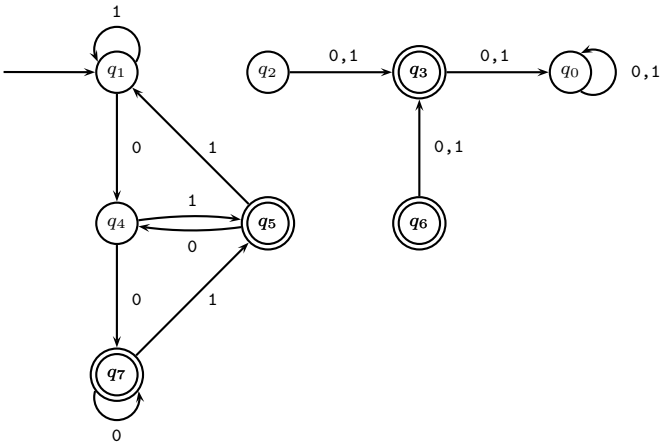


Abbildung 3.10: Zustandsdiagramm des DFA aus der Potenzmengen–Konstruktion

---

$\Longrightarrow$ Übung [3.37]

---

(3) $\Longrightarrow$ (1).

Konstruktion einer regulären Grammatik aus einem DFA.

Wir müssen zeigen: Gegeben ist ein deterministischer endlicher Automat M, der die Sprache L akzeptiert, *i.e.* $L = L(M)$. Wir können eine reguläre Grammatik G angeben, die die Sprache L generiert, *i.e.* $L = L(G)$.

Sei also $M = (Q, \Sigma, \delta, q^s, F)$ ein `DFA` mit $L = L(M)$. Betrachte die Grammatik

$$G = (N, T, P, S)$$

mit

- Die Menge der Variablen N ist:

  $$N = \{X_q, q \in Q\},$$

  i.e. jedem Zustand des `DFA` M wird eine Variable X_q zugeordnet.

- Die Menge der Terminalsymbolen T ist gleich dem Eingabealphabet des Automaten M, *i.e.*

  $$T = \Sigma.$$

- Die Menge der Produktionen der Grammatik ergibt sich durch die Übergangsfunktion des Automaten. Wenn der Automat die Übergangsfunktion $\delta(q, a) = p$ hat, ist die Regel $X_q \longrightarrow aX_p$ hinzuzufügen. Zusätzlich werden noch Regeln $X_q \longrightarrow \lambda$ benötigt für alle Acceptzustände $q \in F$. Damit:

  $$P = \{X_q \longrightarrow aX_p, \delta(q, a) = p\} \cup \{X_q \longrightarrow \lambda,\ \forall\, q \in F\}$$

- Die Startvariable S der Grammatik G ist

  $$S = X_{q^s}.$$

Damit gilt: Für alle $w \in \Sigma^*$ und alle $q \in Q$:

$$S \overset{|w|}{\underset{G}{\Longrightarrow}} wX_q \iff \delta(q_s, w) = q. \tag{3.4}$$

(Beweis durch Induktion über die Länge von w). Es folgt dann:

$$\begin{aligned} w \in L &\iff \delta(q_s, w) = q \quad \text{für ein } q \in F \\ &\overset{(3.4)}{\iff} S \overset{|w|}{\underset{G}{\Longrightarrow}} wX_q \quad \text{für ein } q \in F \\ &\iff S \overset{|w|+1}{\underset{G}{\Longrightarrow}} w, \end{aligned}$$

wobei die letzte Äquivalenz aus der Wahl von P folgt.

Beispiel [3.43]

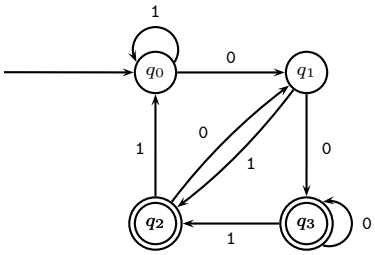
Wir betrachten den DFA aus dem Beispiel [3.42], der aus der Potenzmengen Konstruktion entwickelt wurde. Es ist

$$M = (Q, \Sigma, \delta, q^s, F)$$

mit

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
- $q^s = q_0$
- $F = \{q_2, q_3\}$
- und der Übergangsfunktion δ:

	0	1
q_0	q_1	q_0
q_1	q_3	q_2
q_2	q_1	q_0
q_3	q_3	q_2



Hieraus erhalten wir die reguläre Grammatik wie folgt:

$$G = (N.T, P, S)$$

mit

- der Menge der Variablen

  $$N = \{Z_0, Z_1, Z_2, Z_3\},$$

  i.e. jedem Zustand $q_i \in Q, i = 0, 1, 2, 3$ des DFA wird eine Variable $Z_i \in N.i = 0, 1, 2, 3$ zugeordnet.

- $T = \Sigma$.
- Startvariable ist Z_0.

- Die Menge der Produktionen P ist:

$$\begin{array}{lcl} Z_0 & \longrightarrow & \mathtt{0}Z_1, \\ Z_0 & \longrightarrow & \mathtt{1}Z_0, \\ Z_1 & \longrightarrow & \mathtt{0}Z_3, \\ Z_1 & \longrightarrow & \mathtt{1}Z_2, \\ Z_2 & \longrightarrow & \mathtt{0}Z_1, \\ Z_2 & \longrightarrow & \mathtt{1}Z_0, \\ Z_3 & \longrightarrow & \mathtt{0}Z_3, \\ Z_3 & \longrightarrow & \mathtt{1}Z_2, \\ Z_2 & \longrightarrow & \lambda, \\ Z_3 & \longrightarrow & \lambda. \end{array}$$

Man beachte, dass die λ–Produktionen eliminiert werden können.

Dies geschieht nach dem folgenden allgemeinen Verfahren:

1. Für alle Variablen $X \in N$ mit

$$X \overset{*}{\underset{G}{\Longrightarrow}} \lambda$$

und alle Regeln der Form $Z \longrightarrow uX$ füge neue Regeln $Z \longrightarrow u$, $u \in T^*$ hinzu.

2. Streiche alle Regeln der Form $X \longrightarrow \lambda$.
3. Falls die rechtslineare Grammatik G das leere Wort λ erzeugt, füge ein neues Startsymbol S' zu der Menge der Nicht–Terminalen hinzu. Weiterhin füge die beiden Produktionen

$$S' \longrightarrow \lambda \mid S$$

zu P hinzu.

Um die Regel $Z_2 \longrightarrow \lambda$ zu eliminieren, fügen wir die Regeln

$$Z_1 \longrightarrow \mathtt{1} \quad \text{und} \quad Z_3 \longrightarrow \mathtt{1}$$

hinzu, um die Regel $Z_3 \longrightarrow \lambda$ zu eliminieren, fügen wir die Regeln

$$Z_1 \longrightarrow \mathtt{0} \quad \text{und} \quad Z_3 \longrightarrow \mathtt{0}$$

hinzu. Dann erhalten wir die folgende Grammatik:

$$\begin{array}{lcl} Z_0 & \longrightarrow & \mathtt{0}Z_1 \mid \mathtt{1}Z_0, \\ Z_1 & \longrightarrow & \mathtt{0}Z_3 \mid \mathtt{1}Z_2 \mid \mathtt{0} \mid \mathtt{1} \\ Z_2 & \longrightarrow & \mathtt{0}Z_1 \mid \mathtt{1}Z_0 \\ Z_3 & \longrightarrow & \mathtt{0}Z_3 \mid \mathtt{1}Z_2 \mid \mathtt{0} \mid \mathtt{1} \end{array}$$

$\Longrightarrow$ Übung [3.38]

Anmerkungen:

(a) Die Konstruktion einer regulären Grammatik aus einem nichtdeterministischen endlichen Automaten kann in der gleichen Weise ausgeführt werden wie aus einem deterministischen endlichen Automaten.

(b) Bei der Potenzmengenkonstruktion eines DFA aus einem NFA — siehe Beispiel [3.41] — wird die Anzahl der Zustände des DFA üblicherweise sehr groß. Wenn der NFA n Zustände hat, dann resultieren 2^n Zustände für den DFA, denn die Potenzmenge einer n–elementigen Menge hat 2^n Elemente. Wie das Zustandsdiagramm [3.10] zeigt, zerfällt diese Konstruktion in zwei disjunkte Teilgraphen. Der rechte Zweig dieses Diagramms ist vom Startzustand aus nicht erreichbar, daher können diese Zustände unbeachtet bleiben, ohne dass sich die Sprache ändert, die dieser Automat akzeptiert. Dies ist ein allgemeiner Sachverhalt:

> Jeder DFA akzeptiert genau eine Sprache. Umgekehrt: Es gibt beliebig viele DFA, die eine vorgegebene reguläre Sprache akzeptieren.

(c) Durch die Aneinanderreihung der ersten beiden vorgestellten Algorithmen ist es möglich, zu einer regulären Grammatik einen DFA für die Erkennung der zugehörigen Sprache zu konstruieren. Auf diese Weise könnte die automatische Generierung eines Scanners für eine Sprache realisiert werden, die mit Hilfe einer regulären Grammatik beschrieben ist.

3.7 Konstruktion von DFA mit Hilfe von Zustandsbäumen

Wir wollen anhand eines Beispiels aus dem Bereich des String search zeigen, wie man die Konstruktion eines deterministischen endliche Automaten aus dem letzten Beweis effizienter gestalten kann. Die Aufgabe bei dem Beispiel besteht darin, einen Textteil in einem vorgegebenen Text zu suchen. Jeder moderne Editor verfügt heute über eine solche Funktionalität.

Beispiel [3.44]

Vorgegeben ist das Alphabet

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

Es liegt ein Text mit Zeichen aus Σ vor, in dem der Textteil *ababc* gesucht werden soll.

Ziel ist es, einen DFA zu entwickeln, der alle Wörter über Σ erkennt, die mit dem gesuchten Textteil enden. Also, die Sprache, die erkannt werden soll, ist

$$L = \{w \in \{a, b, c\}^* \mid w = w'ababc, w' \in \Sigma^*\}. \tag{3.5}$$

Es ist nicht schwer, für die gegebene Sprache einen NFA anzugeben. Das Zustandsdiagramm dieses NFA ist in der Abbildung [3.11] dargestellt.

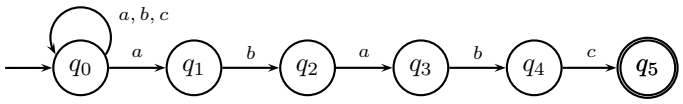


Abbildung 3.11: Zustandsdiagramm des NFA, der die Sprache (3.5) erkennt.

Formal können wir diesen NFA folgendermaßen angeben:

$$N = (Q, \Sigma, \delta, q_s, F)$$

mit

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3, q_4, q_5\} \\
\Sigma &= \{a, b, c\}, \\
q^s &= q_0, \\
F &= \{q_5\}
\end{aligned}
$$

und der Übergangsfunktion δ:

	a	b	c
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\emptyset$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\emptyset$	$\emptyset$	$\{q_5\}$
q_4	$\emptyset$	$\emptyset$	$\emptyset$

Der **Zustandsbaum** für diesen NFA entsteht dadurch, dass man nur diejenigen Zustände des DFA konstruiert, die von dem neuen Anfangszustand aus über einen verbundenen Weg von gerichteten Kanten erreichbar sind.

Dazu beginnt man mit dem neuen Anfangszustand und verzweigt zu allen direkten Nachfolgern. Dies wiederholt man mit jedem der so erhaltenen neuen Zustände. Hat man für einen Zustand schon einmal alle direkten Nachfolger hinzugefügt, so braucht dieser nicht mehr weiter berücksichtigt werden.

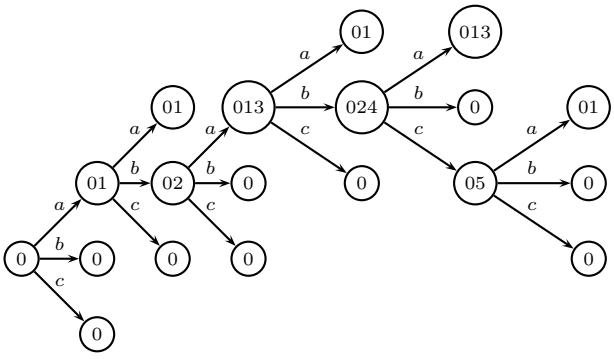


Abbildung 3.12: Zustandsbaum

Damit haben wir für alle Zustände, die von dem neuen Anfangszustand erreichbar sind, die entsprechenden Übergänge konstruiert und können nun den deterministischen endliche Automaten angeben.

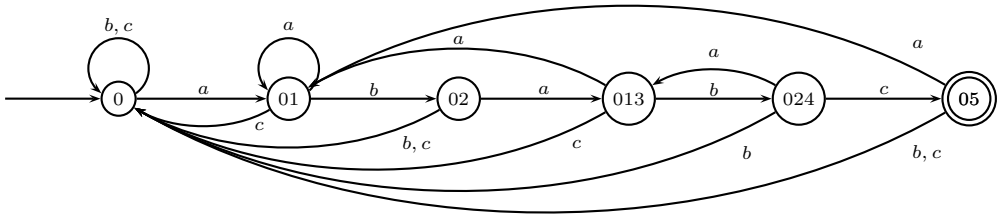


Abbildung 3.13: DFA

Der Übersichtlichkeit halber geben wir den DFA in der ursprünglichen Notation der Zustände an. Die formale Beschreibung des erhaltenen DFA lautet wie folgt:

$$M = (Q', \Sigma, \delta', q^{s'}, F')$$

mit den sechs Zuständen

$$Q' = \{\{q_0\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_0, q_5\}, \{q_0, q_1, q_3\}, \{q_0, q_2, q_4\}\} \subset \wp(Q),$$

dem Alphabet

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

dem Startzustand

$$q^{s'} = \{q_0\},$$

der Menge der Acceptzustände

$$F = \{\{q_0, q_5\}\}$$

und der Übergangsfunktion δ':

	a	b	c
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0\}$
$\{q_0, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0\}$	$\{q_0\}$
$\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_2, q_4\}$	$\{q_0\}$
$\{q_0, q_2, q_4\}$	$\{q_0, q_1, q_3\}$	$\{q_0\}$	$\{q_0, q_5\}$
$\{q_0, q_5\}$	$\{q_0, q_1 i\}$	$\{q_0\}$	$\{q_0\}$

$\Longrightarrow$ Übung [3.26]

Der deterministische endliche Automat zeigt, wie die Textsuche in einem Editor realisiert werden kann.

1. Übersetze den zu suchenden Textteil in einen nichtdeterministischen endlichen Automaten. Dieser hat stets die gleiche einfache Struktur, der Suchstring bestimmt die Folge von Zuständen, die in den Acceptzustand führen.

2. Konstruiere aus dem nichtdeterministsichen endlichen Automaten einen äquivalenten deterministischen endlichen Automaten. Dazu verwendet man das Zustandsbaum–Verfahren.

3. Lese den gesamten Text mit dem deterministischen endlichen Automaten. Dieser befindet sich immer dann in einem Acceptzustand, wenn er ein Vorkommen des gesuchten Textteils gefunden hat. Unterbreche die Verarbeitung des Automaten jeweils im Endzustand, markiere den Textteil und biete dem Benutzer geeignete Interaktionsmöglichkeiten wie *Abbrechen* oder *Weitersuchen* an.

Man kann mit dieser Vorhgehensweise auch problemlos die Suche nach mehreren Textteilen gleichzeitig realisieren, dies zeigt das folgende Beispiel.

Beispiel [3.45]

Wir betrachten wieder das Alphabet $\Sigma = \{a, b, c\}$, gesucht werden die Strings abc, ca und $baba$. Der zugehörige NFA hat das folgende Zustandsdiagramm:

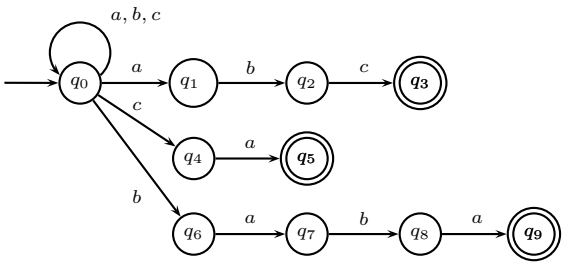


Abbildung 3.14: NFA, der die Strings abc, ca und $baba$ erkennt.

Obwohl die endliche Automaten einen sehr einfachen Weg zeigen, die Suche nach Textteilen durchzuführen, werden in der Praxis effizientere Algorithmen verwendet, beispielsweise den **Rabin–Karp–Algorithmus** oder der **Bayer–Moore–Algorithmus**.

3.8 Minimierung endlicher Automaten

Endliche Automaten werden in vielen Bereichen eingesetzt. Im Allgemeinen hat man es dabei mit sehr umfangreichen und komplexen Automaten zu tun. Daher ist es relevant, ob ein Automat verkleinert werden kann, ohne dass sich die von ihm erkannte Sprache ändert. Werden endliche Automaten zum Beispiel als Schaltwerke in Hardware realisiert, hat dies direkte Auswirkungen auf die Schaltungsgröße.[8]

In diesem Abschnitt sehen wir uns die Minimierung deterministischer endlicher Automaten an. Bei den `DFA` bezieht sich die Minimierung auf die Anzahl der Zustände. Wir betrachten hier den in dem Buch von Hopcroft, Montwani und Ullman betrachteten Algorithmus[9], der auf dem Satz von Myhill und Nerode beruht.

Für die Generierung des zu einem vorgegebenen deterministischen endliche Automaten äquivalenten Minimalautomaten werden zunächst alle Zustände entfernt, die vom Startzustand nicht erreichbar sind. Solche einen `DFA` nennt man einen **vereinfachten Automaten**.. In einem vereinfachten Automaten werden zwei Zustände als k–äquivalent bezeichnet, wenn bei Eingabe aller Wörter der Länge k oder weniger kein Unterschied in ihrem Akzeptanzvehalten vorliegt. Das impliziert, dass der Automat bei Eingabe jedes dieser Wörter entweder aus beiden Zuständen in einen Acceptzustand oder aus beiden in einen Nicht–Acceptzustand übergeht. Der Algorithmus markiert — beginnend mit $k = 0$ — iterativ alle Zustandspaare, die nicht k–äquivalent sind und erhöht k um 1. Wenn in einer Iteration keine Markierung mehr vorgenommen wird, terminiert das Verfahren. Die dann nicht markierten Zustandspaare sind äquivalent. Der **Minimalautomat** wird anschließend durch das Zusammenfassen der äquivalenten Zustände erzeugt. Der Minimalautomat ist bis auf Isomorphie eindeutig.

[8] Zu diesem Themenkreis siehe das Buch von Kohavi und Jha, [35].
[9] Siehe [33], pp. 154 – 161, [34], Kap. 3.4, [37], Lecture 14, [40], Chapt. 2.6.

Markierungsalgorithmus

INPUT:
Einen vereinfachten DFA $M = (Q, \Sigma, \delta, s, F)$.

INITIALISIERUNG:
Erstelle eine Tabelle mit allen Paaren $\{p, q\}, p, q \in Q$.
Markiere in der Tabelle alle Paare mit $p \in F \wedge q \notin F$.

ITERATION:
Wiederhole die folgenden Schritte:
Falls es ein nicht markiertes Paar $\{p, q\}$ gibt, so dass das Paar

$$\big\{\delta(p, a), \delta(q, a)\big\}$$

markiert ist für irgend ein $a \in \Sigma$, dann markiere $\{p, q\}$.
Wenn keine Änderungen mehr auftreten, beende die Iteration.

TERMINIERUNG
Ist die Schleife terminiert, STOP,

$$p \sim_D q \iff \{p, q\} \text{ ist nicht markiert.}$$

Anmerkungen:

❶ Wir müssen in der Regel das gleiche Paar $\{p, q\}$ mehrmals in Schritt 3 untersuchen, denn jede Änderung der Tabelle kann nach sich ziehen, dass $\{p, q\}$ zu markieren ist. Das Verfahren terminiert nur dann, wenn ein kompletter Durchgang der Tabelle zu keiner Änderung mehr führt.

❷ Der Algorithmus läuft lediglich eine endliche Anzahl von Schritten, denn es gibt lediglich $\binom{n}{2}$ mögliche Markierungen, die man machen kann.

❸ Schritt 4 ist die Korrektheitsaussage des Algorithmus, die zu beweisen ist.

❹ Dieses Verfahren zu Minimierung die Anzahl der Zustände funktioniert nur mit deterministischen endliche Automaten und ist auf nichtdeterministische Automatemn nicht anwendbar.

Beispiel [3.46]

In diesem Beispiel betrachten wir die beiden DFA aus der Abbildung [3.15].
Beide Automaten akzeptieren die Menge

$$\big\{a, b\big\} \cup \big\{ \text{ Strings der Länge 3 oder größer}\big\}.$$

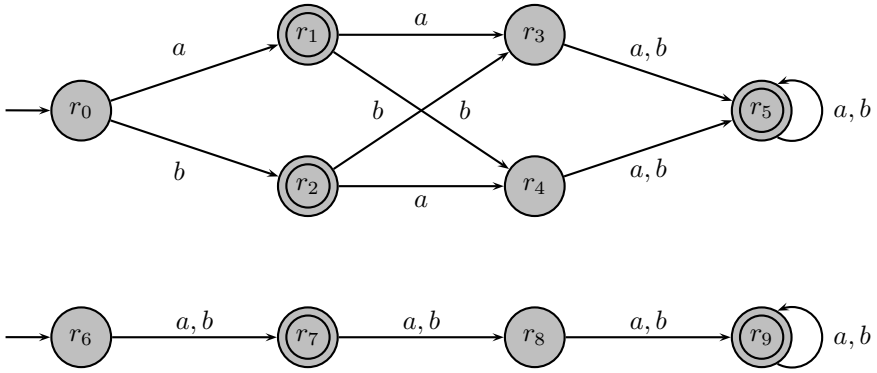


Abbildung 3.15: Zwei äquivalente Automaten, die die Sprache $L = \{a, b\} \cup$ { Strings der Länge 3 oder größer } akzeptieren.

Im ersten Automaten sind die beiden Zustände r_3 und r_4 äquivalent, denn beide gehen in den Zustand r_5 bei beiden Inputsymbolen über. Daher besteht kein Grund, diese beiden Zustände zu separieren. Sind diese beiden Zustände zu einem Zustand zusammengefasst, können wir mit der gleichen Argumentation auch die beiden Zustände r_1 und r_2 zusammenziehen, was auf den unteren Automaten führt. Der Zustand r_0 wird zu r_6, die beiden Zustände r_1, r_2 werden zu r_7, r_3 und r_4 werden zu r_8 zusammengezogen, und Zustand r_5 wird zu Zustand r_9.

Wir wenden den Markierungsalgorithmus um den oberen Automaten der Abbildung [3.15] zu minimieren. Die Übergangsfunktion des oberen Automaten der Abbildung [3.15] lautet (in der unten stehenden Tabelle haben wir alle Accept–Zustände mit einem F markiert):

δ	a	b
r_0	r_1	r_2
r_1 F	r_3	r_4
r_2 F	r_4	r_3
r_3	r_5	r_5
r_4	r_5	r_5
r_5 F	r_5	r_5

Nach Schritt 1 des Algorithmus erstellen wir eine Tabelle, wobei erst mal keine Markierungen vorgenommen werden:

r_0					
–	r_1				
–	–	r_2			
–	–	–	r_3		
–	–	–	–	r_4	
–	–	–	–	–	r_5

Schritt 2: Markieren aller Paare $\{p, q\}$, wobei $p \in F$ und $q \notin F$, also alle Paare von Zuständen, wobei der eine Zustand ein Accept–Zustand ist, der andere nicht. Da die Zustände r_1, r_2 und r_5 Accept-Zustände sind, erhalten wir folgende Tabelle:

r_0					
X	r_1				
X	–	r_2			
–	X	X	r_3		
–	X	X	–	r_4	
X	–	–	X	X	r_5

In dem nun folgenden Schritt müssen wir alle nicht–markierten Einträge — diese nennen wir $\{p, q\}$ — der obigen Tabelle untersuchen und nachprüfen, ob der Zustand $\big\{\delta(p, x), \delta(q, x)\big\}$ mit $x \in \Sigma$ ein markierte Zustand ist oder nicht. Betrachte also das Paar r_0, r_3, das in obiger Tabelle nicht markiert ist. Unter dem Input a geht der Zustand r_0 in den Zustand r_1 über und r_3 nach r_5, *i.e.*:

$$\{r_0, r_3\} \xrightarrow{a} \{r_1, r_5\}.$$

Unter dem Input b haben wir:

$$\{r_0, r_3\} \xrightarrow{b} \{r_2, r_5\}.$$

Nun sind weder das Zustandspaar $\{r_1, r_5\}$ noch $\{r_2, r_5\}$ markiert, also wird auch $\{r_0, r_1\}$ (noch) nicht markiert. Die Analyse der noch nicht markierten Zustandspaare in obiger Tabelle nach diesem Verfahren ergibt:

$\{r_0, r_4\}$	$\xrightarrow{a}$	$\{r_1, r_5\}$		nicht markiert
$\{r_0, r_4\}$	$\xrightarrow{b}$	$\{r_2, r_5\}$		nicht markiert
$\{r_1, r_2\}$	$\xrightarrow{a}$	$\{r_3, r_4\}$		nicht markiert
$\{r_1, r_2\}$	$\xrightarrow{b}$	$\{r_4, r_3\}$	$= \{r_3, r_4\}$	nicht markiert
$\{r_3, r_4\}$	$\xrightarrow{a}$	$\{r_5, r_5\}$		nicht markiert
$\{r_3, r_4\}$	$\xrightarrow{b}$	$\{r_5, r_5\}$		nicht markiert
$\{r_1, r_5\}$	$\xrightarrow{a}$	$\{r_3, r_5\}$		markiert
$\{r_1, r_5\}$	$\xrightarrow{b}$	$\{r_5, r_5\}$		nicht markiert
$\{r_2, r_5\}$	$\xrightarrow{a}$	$\{r_4, r_5\}$		markiert
$\{r_2, r_5\}$	$\xrightarrow{b}$	$\{r_5, r_5\}$		nicht markiert

Damit gehen also die beiden Paare $\{r_1, r_5\}$ und $\{r_2, r_5\}$ in markierte Zustandspaare über, daher markieren wir diese beiden Paare ebenfalls. Daher ergibt sich nach einem ersten Durchlauf:

r_0					
X	r_1				
X	–	r_2			
–	X	X	r_3		
–	X	X	–	r_4	
X	X	X	X	X	r_5

Nun gehen wir nochmals durch die Tabelle und analysieren die noch nicht markierten Zustandpaare erneut:

$$\begin{array}{lcll}
\{r_1, r_2\} & \xrightarrow{a} & \{r_3, r_4\} & \text{nicht markiert} \\
\{r_1, r_2\} & \xrightarrow{b} & \{r_4, r_3\} = \{r_3, r_4\} & \text{nicht markiert} \\
\{r_0, r_3\} & \xrightarrow{a} & \{r_1, r_5\} & \text{markiert} \\
\{r_0, r_3\} & \xrightarrow{b} & \{r_2, r_5\} & \text{markiert} \\
\{r_0, r_4\} & \xrightarrow{a} & \{r_1, r_5\} & \text{markiert} \\
\{r_0, r_4\} & \xrightarrow{b} & \{r_2, r_5\} & \text{markiert} \\
\{r_3, r_4\} & \xrightarrow{b} & \{r_5, r_5\} & \text{nicht markiert} \\
\{r_3, r_4\} & \xrightarrow{a} & \{r_5, r_5\} & \text{nicht markiert}
\end{array}$$

Damit sind nach dieser Runde die beiden Paare $\{r_0, r_3\}$ und $\{r_0, r_4\}$ ebenfalls zu markieren. Dies ergibt folgende Tabelle

r_0					
X	r_1				
X	–	r_2			
X	X	X	r_3		
X	X	X	–	r_4	
X	X	X	X	X	r_5

Wir gehen diese Tabelle erneut durch und untersuchen die Zustandsänderungen der beiden noch nicht markierten Paare $\{r_1, r_2\}$ und $\{r_3, r_4\}$. Da

$$\{r_1, r_2\} \xrightarrow{a,b} \{r_3, r_4\}$$

und

$$\{r_3, r_4\} \xrightarrow{a,b} \{r_5, r_5\}$$

und beide Zustandspaare sind nicht markiert – das Paar $\{r_5, r_5\}$ taucht in keiner Tabelle auf und wird daher niemals markiert – wird kein weiterer Zustand mehr markiert und der Algorithmus terminiert.

Das Verfahren liefert also zwei unmarkierte Paare $\{r_1, r_2\}$ und $\{r_3, r_4\}$, und wir folgern:

$$r_1 \sim_D r_2 \qquad r_3 \sim_D r_4.$$

$\Longrightarrow$ Übung [3.41]

Beispiel [3.47]

Das folgende Beispiel ([33, pp. 163]) zeigt, dass das Verfahren, auf dem der Table–filling Algorithmus beruht, auf `NFA`s nicht anwendbar ist. Die Abbildung [3.16] zeigt das Zustandsdiagramm eines `NFA`. Kein Zustände sind äquivalent.

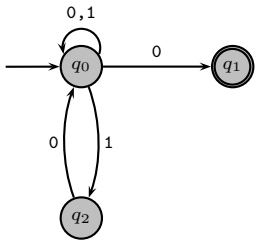


Abbildung 3.16: `NFA` mit drei nicht äquivalenten Zuständen.

Der Acceptzustand q_1 kann sicherlich von den beiden anderen Zuständen q_0 und q_1 unterschieden werden, *i.e.* q_1 hat beispielsweise keinen Folgezustand, während dies bei q_0, q_1 zutrifft. Die Zustände q_0 und q_2 sind ebenfalls nicht äquivalent. Der Folgezustand von q_2 ist q_0, was kein Acceptzustand ist, während die Folgezustände von q_0 die Zustände $\{q_0, q_1\}$ sind bei der Verarbeitung der `0`, was einen Acceptzustand beinhaltet. Daher führt die Gruppierung äquivalenter Zustände nicht zu einer Reduktion der Anzahl der Zustände.

Betrachte den `NFA` in der Abbildung [3.17].

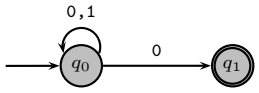


Abbildung 3.17: `NFA`, der alle Bitstrings akzeptiert, die mit `0` enden.

Die Abbildung zeigt das Zustandsdiagramm eines `NFA`, der die gleiche Sprache akzeptiert mit nur zwei Zuständen.

3.9 Abschlusseigenschaften regulärer Sprachen

Sei $\mathbb{N} = \{0, 1, 2, 3, ...\}$ die Menge der natürlichen Zahlen. Wenn man sagt, dass die Menge $\mathbb{N}$ *abgeschlossen* unter der Multiplikation oder Addition ist, dann bedeutet das, dass $\forall x, y \in \mathbb{N}$ gilt $x \times y \in \mathbb{N}$ oder $x + y \in \mathbb{N}$. Im Gegensatz dazu ist die Menge der natürlichen Zahlen $\mathbb{N}$ nicht abgeschlossen bezüglich der Divisionsoperation, denn x und y liegen zwar in $\mathbb{N}$, jedoch die rationale Zahl x/y im allgemeinen nicht. Allgemein gilt, dass eine Menge von Objekten unter einer Operation abgeschlossen ist, wenn das aus der Operation hervorgehende Objekt ebenfalls in der Menge liegt.

Wir zeigen in diesem Kapitel, dass die Menge der regulären Sprachen unter den Operationen Vereinigung, Komplement, Durchschnitt, Konkatenation und Kleene–Stern abgeschlossen ist. Wir beginnen mit den beiden Mengenoperationen Vereinigung und Durchschnitt.

3.9.1 Vereinigung und Durchschnitt

Theorem [3.10]:

(a) Die Klasse der regulären Sprachen ist unter der Vereinigung abgeschlossen. Sind L_1 und L_2 reguläre Sprachen, dann ist $L = L_1 \cup L_2$ ebenfalls eine reguläre Sprache.

(b) Die Klasse der regulären Sprachen ist unter der Durchschnittsbildung abgeschlossen. Sind L_1 und L_2 reguläre Sprachen, dann ist $L = L_1 \cap L_2$ ebenfalls eine reguläre Sprache.

Beweis:

Zunächst die Beweisidee. Wir haben zwei beliebige reguläre Sprachen L_1 und L_2 und müssen zeigen, dass $L = L_1 \cup L_2$ ebenfalls regulär ist. Da L_1 und L_2 beide reguläre Sprachen sind, wissen wir, dass es einen endlichen Automaten M_1 gibt, der die Sprache L_1 erkennt und ebenso einen endlichen Automaten M_2, der die Sprache L_2 erkennt. Um zu zeigen, dass die Vereinigung $L_1 \cup L_2$ ebenfalls regulär ist, müssen wir einen endlichen Automaten finden — wir nennen ihn den Produktautomaten M — der die Sprache $L_1 \cup L_2$ erkennt.

Der eigentliche Beweis ist ein Konstruktionsbeweis, wir können den Produktautomaten M explizit aus den bekannten endlichen Automaten M_1 und M_2 konstruieren. Die Maschine M muss die Eingabe genau dann erkennen, wenn *entweder* M_1 *oder* M_2 die Eingabe akzeptiert. M funktioniert dadurch, dass M beide endlichen Automaten M_1 und M_2 gleichzeitig simuliert und die Zeichenketten akzeptiert, wenn eine der beiden Simulationen den Input akzeptiert.

Seien also die beiden DFA gegeben

$$M_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$
$$M_2 = (Q_2, \Sigma, \delta_2, q_2^s, F_2)$$

wobei M_1 die reguläre Sprache L_1 erkennt und M_2 die reguläre Sprache L_2.

Wir konstruieren nun einen endlichen Automaten M, der die Vereinigungssprache $L_1 \cup L_2$ erkennt. Gleichzeitig können wir auch einen Automaten angeben, der die Durchschnittssprache $L_1 \cap L_2$ akzeptiert. Es ist:

$$M = (Q, \Sigma, \delta, q^s, F)$$

1. $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ und } r_2 \in Q_2\}$
   Dies ist das *kartesische Produkt* der Mengen Q_1 und Q_2, das man in der Form $Q_1 \times Q_2$ schreibt. Diese Menge besteht aus allen Paaren von Zuständen, der erste Zustand ist aus Q_1, der zweite aus Q_2.

2. Σ, das zugrundeliegende Inputalphabet, ist das gleiche wie in den Automaten M_1 und M_2. Der Einfachheit halber nehmen wir dies hier (und in den folgenden Theoremen) an. Liegen unterschiedliche Alphabete vor, operiert man mit $\Sigma = \Sigma_1 \cup \Sigma_2$.

3. Wir definieren die Übergangsfunktion δ folgendermaßen: Für alle Paare $(r_1, r_2) \in Q$ und jedes $a \in \Sigma$ sei

   $$\delta((r_1, r_2), a) := (\delta_1(r_1, a), \delta_2(r_2, a))$$

   Durch diese Definition haben wir erreicht, dass δ einen Zustand von M liest, zusammen mit einem Input–Zeichen und daraus einen Zustand erzeugt, der ebenfalls in M liegt.

4. Der Startzustand q^s ist das Paar $(q_1, q_2) \in Q = Q_1 \times Q_2$.

5. F ist die Menge der Paare, bei der eins der beiden Einträge ein Accept–Zustand von M_1 *oder* M_2 ist. Dies lässt sich schreiben in der Form:

   $$F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ oder } r_2 \in F_2\}$$

   Dieser Ausdruck ist

   $$F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$$

6. Für den Automaten, der die Durchschnittssprache akzeptiert, setzen wir

   $$F = F_1 \times F_2.$$

   F ist die Menge aller Paare, mit $r_1 \in F_1$ **und** $r_2 \in F_2$.

Beispiel [3.48]

In diesem Beispiel[10] betrachten wir

- einen `DFA` M_1, der die Sprache L_1 über dem Alphabet $\Sigma = \{a, b\}$ akzeptiert mit
$$L_1 = \{\omega \in \Sigma^* \mid \omega \text{ enthält den Substring } bb\}$$
- einen zweiten `DFA` M_2, der die Sprache L_2 über dem Alphabet $\Sigma = \{a, b\}$ akzeptiert mit
$$L_2 = \{\omega \in \Sigma^* \mid \omega \text{ enthält } \underline{\text{nicht}} \text{ den Substring } aa\}$$

Ziel ist es nun, unter Verwendung des oben angegebenen Algorithmus einen `DFA` — diesen nennen wir M — zu konstruieren, der die Sprache $L = L_1 \cup L_2$ akzeptiert. Das heißt, der zu konstruierende Automat akzeptiert alle Worte, die entweder den Substring bb enthalten oder nicht den den Substring aa.

Konstruieren wir zunächst die beiden Maschinen M_1 und M_2.

Der `DFA` M_1 kann durch folgendes Quintupel spezifiziert werden:

$$M_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$$

mit

- Die Menge der Zustände $Q_1 = \{q_0, q_1, q_2\}$
- Das Eingabealphabet $\Sigma = \{a, b\}$
- Der Startzustand q_0
- Die Menge der Acceptzustände $F_1 = \{q_2\}$

und die Übergangsfunktion:

δ_1	a	b
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_2	q_2

Das Zustandsdiagramm dieses Automaten ist in der Abbildung [3.18] dargestellt.

Der Automat, der alle Worte über $\Sigma = \{a, b\}$ akzeptiert, die den Substring aa *nicht* enthalten, wird formal beschrieben durch das Quintupel

$$M_2 = (Q_2, \Sigma, \delta_2, p_0, F_2)$$

mit

[10] Siehe Sudkamp [45], Examples 5.3.1, 5.3.2 und 5.3.3.

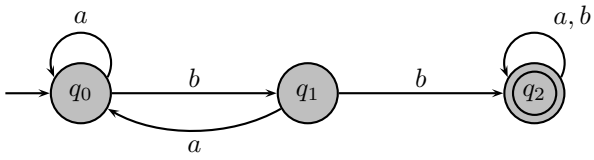


Abbildung 3.18: Ein DFA, der die Menge aller Worte über $\Sigma = \{a, b\}$ akzeptiert, die den Substring bb enthalten

- Die Menge der Zustände $Q_2 = \{p_0, p_1, p_2\}$
- Das Eingabealphabet $\Sigma = \{a, b\}$
- Der Startzustand p_0
- Die Menge der Acceptzustände $F_2 = \{p_0, p_1\}$

und die Übergangsfunktion:

δ_2	a	b
p_0	p_1	p_0
p_1	p_2	p_0
p_2	p_2	p_2

Das Zustandsdiagramm dieses Automaten ist in der Abbildung [3.19] dagestellt.

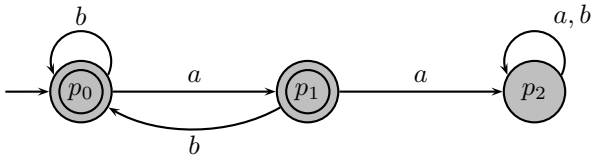


Abbildung 3.19: Zustandsdiagramm des DFA, die alle Strings über $\Sigma = \{a, b\}$ akzeptiert, die den Substring aa nicht enthalten

Ausgehend von den beiden DFA M_1 und M_2 konstruieren wir den DFA der die Sprache

$$L = L_1 \cup L_2$$
$$= \{\omega \in \Sigma^* \mid \omega \text{ enthält den Substring } bb \text{ oder nicht den Substring } aa\ \}$$

Dieser Automat ist ebenfalls beschreibbar durch ein Quintupel

$$M = (Q, \Sigma, \delta, r_0, F)$$

mit

- die Zustände des Automaten ergeben sich aus

$$Q = \big\{(q_i, p_j) \mid q_i \in Q_1, i = 0, 1, 2; p_j \in Q_2, j = 0, 1, 2\big\}.$$

  Damit hat der Automat M neun Zustände, die wir $r_k, k = 0, 1, \ldots, 8$ nennen, diese sind gegeben durch die Paare:

$$\begin{array}{llll} r_0 & = (q_0, p_0) & r_1 & = (q_1, p_0) \\ r_2 & = (q_2, p_0) & r_3 & = (q_0, p_1) \\ r_4 & = (q_1, p_1) & r_5 & = (q_2, p_1) \\ r_6 & = (q_0, p_2) & r_7 & = (q_1, p_2) \\ r_8 & = (q_2, p_2) & & \end{array}$$

- das Input Alphabet ist $\Sigma = \big\{a, b\big\}$.

- der Startzustand ist $r_0 = (q_0, p_0)$.

- die Menge der Acceptzustände ist

$$\begin{aligned} F &= \big\{(q_i, p_j) \in Q_1 \times Q_2 \mid q_i \in F_1 \text{ oder } p_j \in F_2\big\} \\ &= \big\{(q_2, p, 0), (q_2, p_1), (q_2, p_2), (q_0, p_0), (q_1, p_0), (q_0, p_1), (q_1, p_1)\big\} \\ &= \big\{r_0, r_1, r_2, r_3, r_4, r_5, r_8\big\}. \end{aligned}$$

- die Übergangsfunktion ist definiert als

$$\delta((q_i, p_j), x) := (\delta_1(q_i, x), \delta_2(p_j, x))$$

  für alle Paare $(q_i, p_j) \in Q$ und jedes $x \in \Sigma$. Damit

$$
\begin{aligned}
\delta(r_0, a) &= \big(\delta_1(q_0, a), \delta_2(p_0, a)\big) = (q_0, p_1) = r_3 \\
\delta(r_0, b) &= \big(\delta_1(q_0, b), \delta_2(p_0, b)\big) = (q_1, p_0) = r_1 \\
\delta(r_1, a) &= \big(\delta_1(q_1, a), \delta_2(p_0, a)\big) = (q_0, p_1) = r_3 \\
\delta(r_1, b) &= \big(\delta_1(q_1, b), \delta_2(p_0, b)\big) = (q_2, p_0) = r_2 \\
\delta(r_2, a) &= \big(\delta_1(q_2, a), \delta_2(p_0, a)\big) = (q_2, p_1) = r_5 \\
\delta(r_2, b) &= \big(\delta_1(q_2, b), \delta_2(p_0, b)\big) = (q_2, p_0) = r_2 \\
\delta(r_3, a) &= \big(\delta_1(q_0, a), \delta_2(p_1, a)\big) = (q_0, p_2) = r_6 \\
\delta(r_3, b) &= \big(\delta_1(q_0, b), \delta_2(p_1, b)\big) = (q_1, p_0) = r_1 \\
\delta(r_4, a) &= \big(\delta_1(q_1, a), \delta_2(p_1, a)\big) = (q_0, p_2) = r_6 \\
\delta(r_4, b) &= \big(\delta_1(q_1, b), \delta_2(p_1, b)\big) = (q_2, p_2) = r_8 \\
\delta(r_5, a) &= \big(\delta_1(q_2, a), \delta_2(p_1, a)\big) = (q_2, p_2) = r_8 \\
\delta(r_5, b) &= \big(\delta_1(q_2, b), \delta_2(p_1, b)\big) = (q_2, p_0) = r_2 \\
\delta(r_6, a) &= \big(\delta_1(q_0, a), \delta_2(p_2, a)\big) = (q_0, p_2) = r_6 \\
\delta(r_6, b) &= \big(\delta_1(q_0, b), \delta_2(p_2, b)\big) = (q_1, p_2) = r_7 \\
\delta(r_7, a) &= \big(\delta_1(q_1, a), \delta_2(p_2, a)\big) = (q_0, p_2) = r_6 \\
\delta(r_7, b) &= \big(\delta_1(q_1, b), \delta_2(p_2, b)\big) = (q_2, p_2) = r_8 \\
\delta(r_8, a) &= \big(\delta_1(q_2, a), \delta_2(p_2, a)\big) = (q_2, p_2) = r_8 \\
\delta(r_8, b) &= \big(\delta_1(q_2, b), \delta_2(p_2, b)\big) = (q_2, p_2) = r_8
\end{aligned}
$$

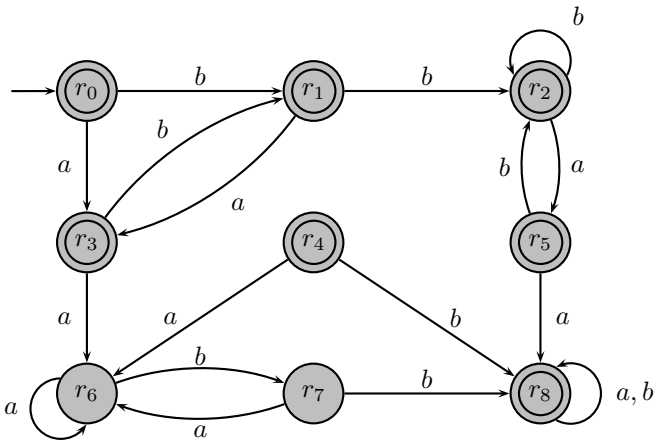


Abbildung 3.20: Der DFA, der die Vereinigungssprache $L = L_1 \cup L_2$ akzeptiert.

$\Longrightarrow$ Übung [3.40]

3.9.2 Komplementbildung

Theorem [3.11]:

Die Klasse der regulären Sprachen ist unter der Komplementbildung abgeschlossen. Das heisst, ist L eine reguläre Sprache, dann ist die komplementäre Sprache $\overline{L} = \Sigma^* \setminus L$ ebenfalls eine reguläre Sprache.

Beweis:

Die Sprache L sei $L(M)$ für einen deterministischen endlichen Automaten M mit

$$M = (Q, \Sigma_1, \delta, q_0, F)$$

und es gelte $L \subseteq \Sigma^*$. Wir nehmen an, dass $\Sigma_1 = \Sigma$ gilt. Denn wenn es Symbole in Σ_1 gibt, die nicht in Σ sind, können alle Übergänge von M auf Symbolen, die nicht in Σ sind, gelöscht werden. Die Tatsache, dass $L \subseteq \Sigma^*$ ist, hat zur Folge, dass die von M erkannte Sprache nicht geändert wird. Wenn es Symbole in Σ gibt, die nicht in Σ_1 liegen, dann taucht keines dieser Symbole in Wörtern der Sprache L auf. Wir können daher einen 'toten' Zustand d in M definieren mit: $\delta(d, a) = d$ für alle a aus Σ und $\delta(q, a) = d$ für alle $q \in Q$ und $a \in \Sigma \setminus \Sigma_1$.

Um nun $\Sigma^* \setminus L$ zu akzeptieren, bildet man das Komplement der Endzustände von M. Es sei also

$$M' = (Q, \Sigma_1, \delta, q_0, Q \setminus F).$$

M' akzeptiert genau dann ein Wort w, wenn $\hat{\delta}(q_0, w)$ in der Menge $Q \setminus F$ liegt, *i.e.* wenn w in $\Sigma^* \setminus L$ liegt.

Beispiel [3.49]

Wir betrachten einen deterministischen, endlichen Automaten M, der die Sprache

$$L = \{aba, abb\}$$

über dem Alphabet $\Sigma = \{a, b\}$ akzeptiert. Diese Sprache besteht also aus genau diesen beiden Worten. Um den `DFA` zu konstruieren, überlegen wir uns, wie er arbeiten muss:

- Wenn als erstes Zeichen ein b gelesen wird, muss der `DFA` vom Startzustand – diesen nennen wir q_0 – in einen Zustand übergehen, der als Garbage dient, also alles aufsammelt, was nicht akzeptiert wird. Diesen Zustand nennen wir q_4. Der Automat bleibt dann in diesem Zustand, wenn irgendwelche weiteren Zeichen gelesen werden.
- Wird als erstes Zeichen ein a gelesen, dann hat man einen potentiellen Kondidaten, der akzeptiert werden muss. Daher geht der `DFA` vom Startzustand q_0 in einen weiteren Zustand q_1 über.
- Ist der Automat im Zustand q_1 und liest ein a, dann muss er in den Garbage verzweigen, denn akzeptiert werden nur Worte, die an zweiter Stelle ein b-Symbol haben. Liest er ein b-Zeichen, dann haben wir einen Kandidaten für ein zu akzeptierendes Wort, der Automat geht in einen weiteren Zustand q_2 über.
- Ist der `DFA` in diesem Zustand, ist es gleichgültig, welches Zeichen gelesen wird, er geht dann in den Acceptzustand — den nennen wir q_3 — über.
- Aus dem Accept–Zustand laufen nun ein a und ein b Übergang in den Garbage–Zustand, denn wenn weitere Zeichen gelesen werden, ist das Wort nicht zu akzeptieren.

Formalisieren wir diesen `DFA`: Formal ist

$$M = \big(Q, \Sigma, \delta, q^s, F\big)$$

mit

- Menge der Zustände $Q = \big\{q_0, q_1, q_2, q_3, q_4\big\}$
- Alphabet: $\Sigma = \big\{a, b\big\}$
- Startzustand q_0
- Acceptzustand $F = \{q_3\}$.

und die Übergangsfunktion δ:

	a	b
q_0	q_1	q_4
q_1	q_4	q_2
q_2	q_3	q_3
q_3	q_4	q_4
q_4	q_4	q_4

Ein `DFA`, der alle Strings akzeptiert ausser aba und abb ist in der Abbildung [3.22] dargestellt.

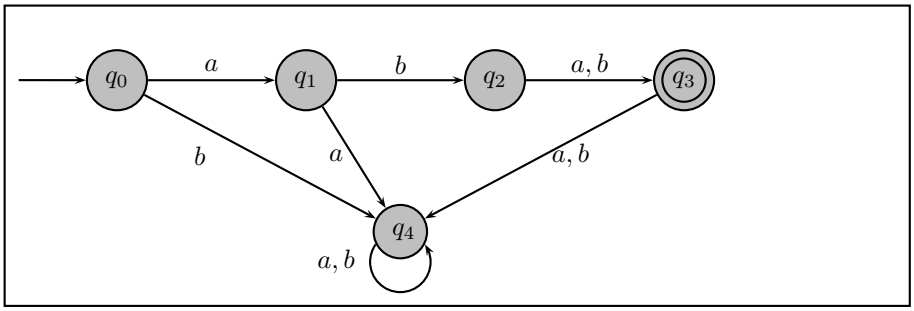


Abbildung 3.21: Der Automat M

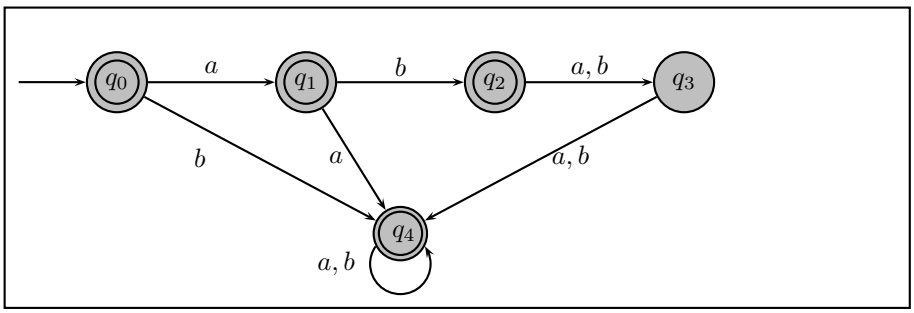


Abbildung 3.22: Der Automat, der die Sprache $\overline{L} = \overline{\{aba, abb\}}$ akzeptiert

---

Beispiel [3.50]

Sei $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. Sei M_1 der deterministische endliche Automat, der die Sprache

$$L_1 = \{w \in \Sigma^* \mid w \text{ endet mit einer } \mathtt{1}\}$$

erkennt und M_2 der Automat, der die Sprache

$$L_2 = \{w \in \Sigma^* \mid w \text{ beginnt mit einer } \mathtt{0}\}$$

erkennt. Wir konstruieren nun mit Hilfe des Verfahrens aus dem Satz [3.10] einen deterministischen endlichen Automaten M, der die Sprache

$$\begin{aligned} L &= L_1 \cap L_2 \\ &= \{w \in \Sigma^* \mid w \text{ beginnt mit einer } \mathtt{0} \textit{ und} \text{ endet mit } \mathtt{1}\} \end{aligned}$$

erkennt.

Der Automat M_1 ist formal gegeben durch

$$M_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$

mit

1. den Zuständen $Q_1 = \{q_1, q_2\}$
2. dem Inputalphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
3. dem Startzustand $q_1^s = q_1$
4. der Menge der Endzustände $F = \{q_2\} \subseteq Q_1$
5. der Übergangsfunktion δ_1 mit

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

Den Automaten M_2 können wir formal beschreiben durch

$$M_2 = (Q_2, \Sigma, \delta_2, q_2^s, F_2)$$

mit

1. den drei Zuständen $Q_2 = \{r_0, r_1, r_2\}$
2. dem Inputalphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
3. dem Startzustand $q_2^s = r_0$
4. der Menge der Endzustände $F = \{r_1\} \subseteq Q_2$
5. der Übergangsfunktion δ_2 mit

	0	1
r_0	r_1	r_2
r_1	r_1	r_1
r_2	r_2	r_2

Das Zustandsdiagramm dieses Automaten ist in der Abbildung [3.23] dargestellt.

Der deterministische endliche Automat M, der die Schnittmenge der Sprachen L_1 und L_2 erkennt ist:

$$M = (Q, \Sigma, \delta, q^s, F)$$

mit

1. den sechs Zuständen

$$\begin{aligned} Q &= Q_1 \times Q_2 \\ &= \{q_1, q_2\} \times \{r_0, r_1, r_2\} \\ &= \{(q_1, r_0), (q_1, r_1), (q_1, r_2), (q_2, r_0), (q_2, r_1), (q_2, r_2)\} \end{aligned}$$

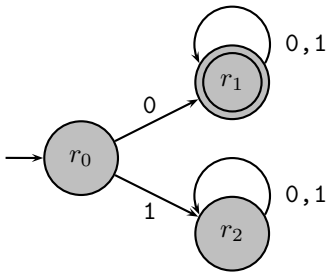


Abbildung 3.23: Zustandsdiagramm des Automaten M_2.

2. dem Inputalphabet $\Sigma = \{0, 1\}$
3. dem Startzustand $q^s = (q_1^s, q_2^s) = (q_1, r_0)$
4. der Menge der Endzustände $F = F_1 \times F_2 = \{(q_2, r_1)\} \subseteq Q_1 \times Q_2$
5. der Übergangsfunktion δ, die wir explizit folgendermaßen konstruieren:

$$
\begin{aligned}
\delta\big((q_1, r_0), 0\big) &= \big(\delta_1(q_1, 0), \delta_2(r_0, 0)\big) = (q_1, r_1)\\
\delta\big((q_1, r_0), 1\big) &= \big(\delta_1(q_1, 1), \delta_2(r_0, 1)\big) = (q_2, r_2)\\
\delta\big((q_2, r_0), 0\big) &= \big(\delta_1(q_2, 0), \delta_2(r_0, 0)\big) = (q_1, r_1)\\
\delta\big((q_2, r_0), 1\big) &= \big(\delta_1(q_2, 1), \delta_2(r_0, 1)\big) = (q_2, r_2)\\
\delta\big((q_1, r_1), 0\big) &= \big(\delta_1(q_1, 0), \delta_2(r_1, 0)\big) = (q_1, r_1)\\
\delta\big((q_1, r_1), 1\big) &= \big(\delta_1(q_1, 1), \delta_2(r_1, 1)\big) = (q_2, r_1)\\
\delta\big((q_2, r_1), 0\big) &= \big(\delta_1(q_2, 0), \delta_2(r_1, 0)\big) = (q_1, r_1)\\
\delta\big((q_2, r_1), 1\big) &= \big(\delta_1(q_2, 1), \delta_2(r_1, 1)\big) = (q_2, r_1)\\
\delta\big((q_1, r_2), 0\big) &= \big(\delta_1(q_1, 0), \delta_2(r_2, 0)\big) = (q_1, r_2)\\
\delta\big((q_1, r_2), 1\big) &= \big(\delta_1(q_1, 1), \delta_2(r_2, 1)\big) = (q_2, r_2)\\
\delta\big((q_2, r_2), 0\big) &= \big(\delta_1(q_2, 0), \delta_2(r_2, 0)\big) = (q_1, r_2)\\
\delta\big((q_2, r_2), 1\big) &= \big(\delta_1(q_2, 1), \delta_2(r_2, 0)\big) = (q_2, r_2)
\end{aligned}
$$

Das Zustandsdiagramm dieses Produktautomaten ist in der Abbildung [3.24] dargestellt. Hier lassen sich eine Reihe von Zuständen eliminieren, ohne daß sich die erkannte Sprache ändert. Wie in der Abbildung [3.24] leicht zu erkennen ist, kann der Zustand (q_2, r_0) nicht vom Startzustand (q_1, r_0) aus erreicht werden. Außerdem sind die beiden Zustände rechts außen, (q_1, r_2) und (q_2, r_2) äquivalent und können zu einem einzigen Zustand kollabieren. Sobald die Maschine in der Zustand (q_2, r_2) gelangt, erreicht sie niemals mehr den Acceptzustand. Sie kann dann nur noch zwischen den Zuständen (q_2, r_2) und (q_1, r_2) wechseln. Daher ist der Zustand (q_1, r_2) redundant und kann mit (q_2, r_2) zusammengezogen werden.

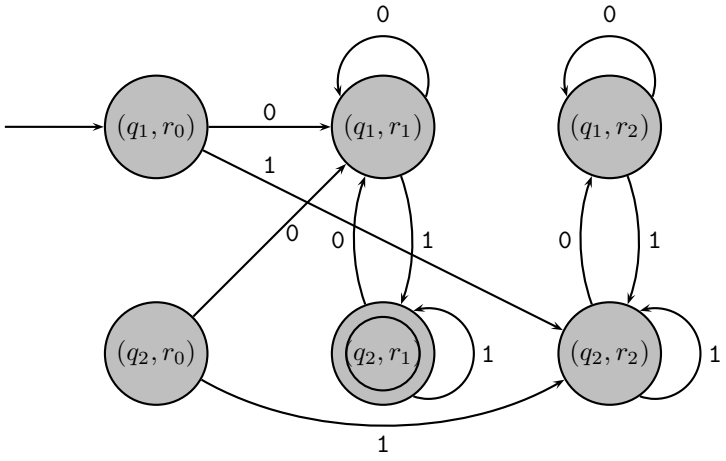


Abbildung 3.24: Das Zustandsdiagramm des Automaten, der die Sprache $L(M) = L_1(M_1) \cap L_2(M_2)$ erkennt.

Es gibt einen Algorithmus, der aus dem sogenannten MYHILL–NERODE Theorem folgt (siehe [34], Kapitel 3.4 oder [37], Lectures 13 – 16), mit dessen Hilfe der sogenannte **Minimalautomat** konstruiert werden kann. Offensichtlich hat der deterministische endliche Automat, der die Sprache

$$L = \{w \in \Sigma^* \mid w \text{ beginnt mit einer } \texttt{0} \textit{ und} \text{ endet mit } \texttt{1}\}$$

vier Zustände, die wir

$$Q = \{q_s, q_0, q_{0x1}, q_{\neg 0}\}$$

nennen wollen mit der Bedeutung:

1. q_s Startzustand, kein Zeichen gelesen
2. q_0 Zustand, in dem genau eine `0` gelesen wurde
3. q_{0x1} Zustand, in dem der String `0x1` gelesen wurde, wobei `x` irgendeine beliebige Folge von `0`en und `1`en ist.
4. $q_{\neg 0}$ Zustand in dem `0` nicht das erste gelesene Zeichen war.

Das Zustandsdiagramm dieses deterministischen endlichen Automaten ist in der Abbildung [3.25] dargestellt.

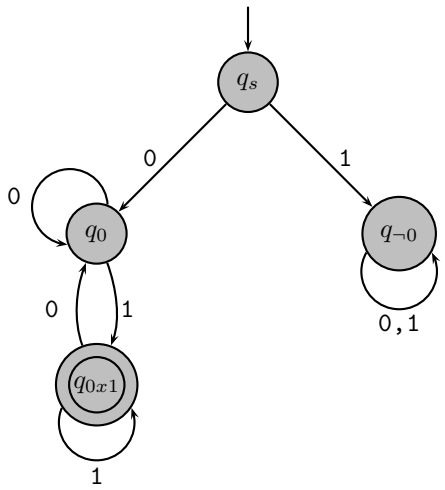


Abbildung 3.25: Das Zustandsdiagramm des Automaten, der die Sprache $L(M) = L_1(M_1) \cap L_2(M_2)$ erkennt.

3.9.3 Konkatenation

Theorem [3.12]:

Die Klasse der regulären Sprachen ist abgeschlossen unter der Konkatenationsoperation.

Beweisidee:

Wir betrachten zwei reguläre Sprachen L_1 und L_2 und zeigen, dass die Konkatenation der beiden Sprachen $L = L_1 \circ L_2$ regulär ist. Die Beweisidee ist, wir nehmen zwei `NFA`s für L_1 und L_2 und kombinieren diese in einen neuen `NFA` N. Diese Konstruktion ist in der Abbildung [3.26] skizziert.

Man ordnet dem Startzustand von N den Startzustand der Maschine N_1 zu. Die Acceptzustände von N_1 erhalten neue λ Übergänge zur Maschine N_2. Diese erlauben der Maschine N_1, in nichtdeterministischer Weise in die Maschine N_2 zu verzweigen, genau dann, wenn sich N_1 in einem Acceptzustand befindet. Dies signalisiert, dass der erste Teil des Inputstrings, der in der Sprache L_1 liegt, bereits akzeptiert ist. Die Acceptzustände von N sind lediglich die Acceptzustände

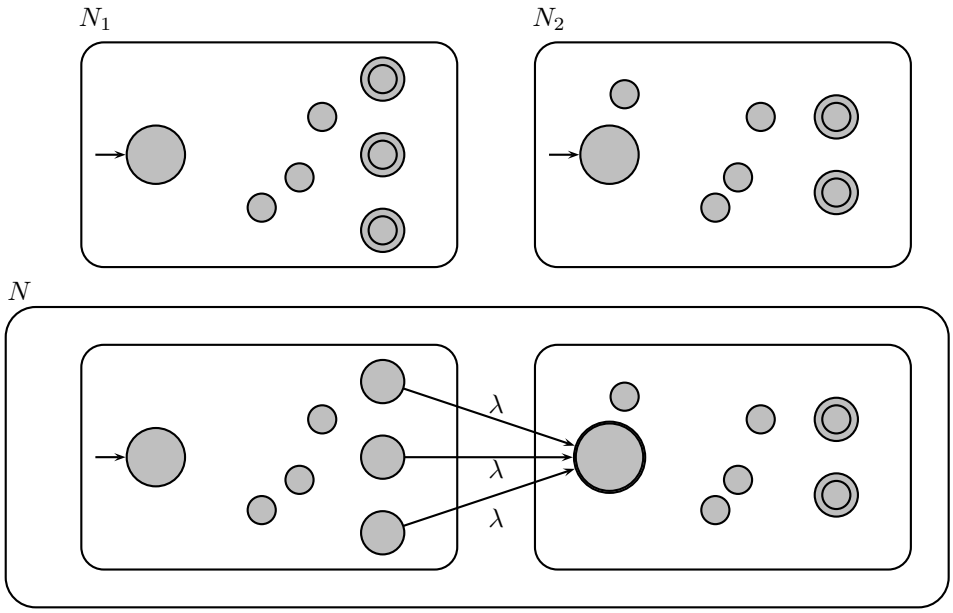


Abbildung 3.26: Konstruktion eines nichtdeterministischen endlichen Automaten N, der die Sprache $L = L_1 \circ L_2$ erkennt.

von N_2. Die Maschine akzeptiert daher einen String genau dann, wenn der Input in zwei Teile aufgeteilt werden kann, der erste Teilstring wird von N_1 akzeptiert, der zweite von N_2. Man kann sich die Arbeitsweise von N so vorstellen, dass die Maschine 'rät', an welcher Stelle im Inputstring die Unterteilung vorzunehmen ist.

Beweis:

Seien

$$N_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$
$$N_2 = (Q_2, \Sigma, \delta_2, q_2^s, F_2)$$

zwei `NFA`s, die die Sprachen L_1 bzw. L_2 erkennen. Dann erkennt der folgende nichtdeterministische endliche Automat $N = (Q, \Sigma, \delta, q_0^s, F)$ die Sprache $L = L_1 \circ L_2$:

1. $Q = Q_1 \cup Q_2$, *i.e.* die Zustände von N sind alle Zustände von N_1 vereinigt mit allen Zuständen von N_2.

2. Der Startzustand q_0^s von N ist der Startzustand von N_1, *i.e.* q_1^s.

3. Die Acceptzustände sind die Menge $F = F_2$. Die Acceptzustände von N sind also genau die Acceptzustände von N_2.

4. Wir definieren die Übergangsfunktion δ so, dass für alle $q \in Q$ und jedes $a \in \Sigma_\lambda$ gilt:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ und } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ und } a \neq \lambda \\ \delta_1(q,a) \cup \{q_2^s\} & q \in F_1 \text{ und } a = \lambda \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

3.9.4 Kleene–Stern

Theorem [3.13]:

Die Klasse der regulären Sprachen ist abgeschlossen unter der KLEENE Star - Operation.

Beweisidee:

Ziel ist es zu zeigen, wenn L eine reguläre Sprache ist, dann ist auch L^* eine reguläre Sprache. Es ist zweckmäßig, sich nochmals die Definition der KLEENE Star Operation anzusehen:

$$L^* = \{w_1 w_2 \dots w_k \mid k \geq 0 \text{ und } w_i \in L \text{ für alle } i, 0 \leq i \leq k\}$$

mit anderen Worten L^* enthält alle möglichen Konkatenationen, die sich aus Worten der Sprache L bilden lassen.

Wir betrachten einen nichtdeterministischen endlichen Automaten N_1, der die Sprache L erkennt. Diesen Automaten modifizieren wir so, dass er die Sprache L^* erkennt. Dies ist in der Abbildung [3.27] dargestellt. Der resultierende `NFA` N akzeptiert den Input genau dann, wenn dieser in mehrere Teilstrings zerlegt werden kann, wobei jeder Teilstring von N_1 akzeptiert wird.

Wir können N genauso aufbauen wie N_1 mit der zusätzlichen Funktionalität von λ–Übergängen von den Acceptzuständen zurück zum Startzustand. Dadurch hat die Maschine N die Möglichkeit, bei der Verarbeitung eines Strings in den Startzustand zurückzuspringen und einen weiteren Teilstring zu verarbeiten, der von N_1 akzeptiert wird. Zusätzlich muß N_1 so erweitert werden, dass auch der leere String λ akzeptiert wird, denn λ ist Element der Menge L^*. Die einfachste

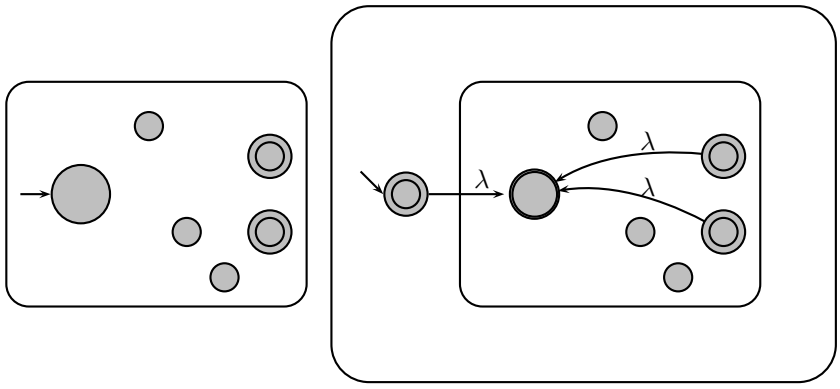


Abbildung 3.27: Konstruktion des nichtdeterministischen endlichen Automaten N, der die Sprache L^* erkennt.

Möglichkeit, dies zu realisieren, ist, aus dem Startzustand von N_1 einen Acceptzustand zu machen. Dies hat aber zur Folge, dass unerwünschte Strings akzeptiert werden. Daher ist die korrekte Erweiterung die Hinzufügung eines neuen Startzustands, der auch Acceptzustand ist und über einen λ Übergang in den alten Startzustand von N_1 verzweigt.

Beweis:

Sei

$$N_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$

ein `NFA`, der die Sprache A erkennt. Der folgende nichtdeterministische endliche Automat $N = (Q, \Sigma, \delta, q_0, F)$ erkennt die Sprache L^*:

1. $Q = \{q_0\} \cup Q_1$, i.e. die Zustände von N sind alle Zustände von N_1 zuzüglich eines neuen Startzustandes.
2. Der Zustand q_0^s ist der neue Startzustand von N.
3. Die Acceptzustände sind $F = \{q_0^s\} \cup F_1$. Die Acceptzustände von N sind also genau die Acceptzustände von N_1 plus den neuen Startzustand.
4. Wir definieren die Übergangsfunktion δ so, dass für alle $q \in Q$ und jedes $a \in \Sigma_\lambda$ gilt:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ und } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ und } a \neq \lambda \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ und } a = \lambda \\ \{q_1\} & q = q_0 \text{ und } a = \lambda \\ \emptyset & q = q_0 \text{ und } a \neq \lambda \end{cases}$$

Anmerkung:

Die Klasse der regulären Sprachen über einem Alphabet Σ enthält die Sprachen $\emptyset, \{\lambda\}, \{\mathtt{a}\}$ für alle $a \in \Sigma$ und wir haben gesehen, dass die Klasse der regulären Sprachen unter anderem abgeschlossen ist unter den drei Operationen

- Vereinigung,
- Konkatenation,
- Kleene Stern.

Wir werden im folgenden Kapitel sehen, dass man aus diesen drei elementaren regulären Sprachen und diesen drei Operationen — diese Operationen nennt man daher auch **reguläre Operationen** — *alle* regulären Sprachen in induktiver Weise erzeugen kann.

Wir wissen beispielsweise, dass die Sprache, die aus dem String `abbac` über dem Alphabet $\Sigma = \{a, b, c\}$ besteht, regulär ist, denn dafür können wir einen `DFA` angeben. Andererseits wissen wir, dass die drei Sprachen, die aus den Wörtern $\{\mathtt{a}\}$, $\{\mathtt{b}\}$ bzw. $\{\mathtt{c}\}$ bestehen, ebenfalls regulär sind. Weiterhin wissen wir, dass die regulären Sprachen abgeschlossen unter der Konkatenationsoperation sind. Daher können wir die reguläre Sprache $\{\mathtt{abbac}\}$ schreiben als $\{\mathtt{a}\}\{\mathtt{b}\}\{\mathtt{b}\}\{\mathtt{a}\}\{\mathtt{c}\}$. Die Frage ist nun, ob dies eine generelle Eigenschaft der regulären Sprachen ist. Das Instrumentarium für diese Untersuchung sind die **regulären Ausdrücke**.

3.10 Grenzen regulärer Sprachen

Die Sprache

$$L = \{w \in \{0,1\}^* \mid \mathtt{w} = 0^{\mathtt{n}}1^{\mathtt{n}}, \mathtt{n} \geq 1\}$$

ist kontextfrei, da wir eine kontextfreie Grammatik G mit den Regeln

$$\begin{array}{lcl} S & \longrightarrow & 0S1 \\ S & \longrightarrow & 01 \end{array}$$

angeben können, die L erzeugt. Wir haben bereits erwähnt, dass diese Sprache nicht regulär ist, *i.e.* es gibt keine reguläre Grammatik, die L erzeugt. Dies werden wir in diesem Abschnitt zeigen.

Theorem [3.14]:

Die Sprache

$$L = \{w \in \{0,1\}^* \mid \mathtt{w} = 0^{\mathtt{n}}1^{\mathtt{n}}, \mathtt{n} \geq 1\} \tag{3.6}$$

ist nicht regulär.

Beweis:

Wir beweisen diesen Satz durch Widerspruch, *i.e.* wir nehmen an, die Sprache L ist regurlär.

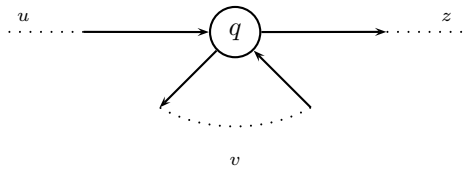
Dann existiert ein NFA N mit $L(N) = L$. Sei k die Anzahl der Zustände von N.

Betrachte das Wort $w = 0^k1^k \in L$.

Der Automat N nimmt bei der Verarbeitung von w der Reihe nach $2k$ Zustände an.[11] Da N nur k Zustände hat, wird bei der Abarbeitung von w mindestens ein Zustand — diesen nennen wir q — mehrfach angenommen. Dazwischen wird ein nichtleerer Teilstring v des Wortes w verarbeitet.

Das bedeutet, die Maschine N liest einen ersten Teil u des Wortes w und gelangt in den Zustand q. Dann durchläuft N einen Loop und ist nach der Verarbeitung von v wieder im Zustand q. Nach der Abarbeitung des Rests z von w gelangt N in einen Acceptzustand.

[11] Der Automat führt eine Pfad aus, der aus einer Folge von Zuständen besteht. Diese Folge von Zuständen beginnt mit dem Startzustand und endet in einem der Acceptzustände.



Der wesentliche Punkt ist nun, dass alle Wörter der Form

$$uv^n z,$$

bei denen die Schleife n–fach durchlaufen wird $n \geq 0$, zu den gleichen Acceptzuständen führen und sind daher ebenfalls Wörter der Sprache, die der DFA akzeptiert. Insbesondere ist also $uvvz \in L(M)$.

Für die Lage des Teilworts v in dem Wort $w = \mathtt{0}^k\mathtt{1}^k$ gibt es drei Möglichkeiten.

Fall 1: $v = \mathtt{0}^j, j \geq 1$.
Der Teilstring besteht nur aus `0`en. Dann muss w die Form haben

$$w = \mathtt{0}^i \underbrace{\mathtt{0}^j}_{v} \mathtt{0}^l\mathtt{1}^k, \quad i + j + l = k$$

Dann hat jedoch das Wort

$$w' = uvvz = \mathtt{0}^i\mathtt{0}^j\mathtt{0}^j\mathtt{0}^l\mathtt{1}^k$$

mehr `0`en als `1`en. Daher ist $w' \notin L(M)$.

Fall 2: $v = \mathtt{0}^j\mathtt{1}^l, j, l \geq 1$.
Der Teilstring besteht aus `0`en und `1`en, *i.e.* w muss die Form haben

$$w = \mathtt{0}^i \underbrace{\mathtt{0}^j\mathtt{1}^l}_{v} \mathtt{1}^m, \quad i + j = k, l + m = k.$$

Dann hat das Wort

$$w' = uvvz = \mathtt{0}^i\mathtt{0}^j\mathtt{1}^l\mathtt{0}^j\mathtt{1}^l\mathtt{1}^m,$$

aber die `0`en und `1`en nicht in der richtigen Reihenfolge. Damit ist w' / $inL(M)$.

Fall 3: $v = \mathtt{1}^j$, dies ist analog zu Fall 1.

In allen drei Fällen ergibt sich ein Widerspruch, daher muss die Annahme, dass L regulär ist, falsch sein. Daher ist die Sprache (3.6) nicht regulär.

Der Beweis des obigen Theorems beruht auf dem sogenannten **Pumping Lemma für reguläre Sprachen**, das folgendermaßen lautet.[12]

Theorem [3.15]:

Pumping Lemma für reguläre Sprachen

Sei L eine reguläre Sprache über dem Alphabet Σ. L wird von einem DFA M akzeptiert. Sei n die Anzahl von Zuständen von M. Dann gilt: Für jedes Wort $w \in L$ mit $\mid w \mid \geq n$ gibt es eine Zerlegung

$$w = uvz$$

mit

1. $\mid uv \mid \leq n$
2. $\mid v \mid > 0$, *i.e.* $v \neq \lambda$,
3. Für alle $i \geq 0$ ist der String uv^iz ebenfalls in L.

Die Sprache (3.6) abstrahiert bei Programmiersprachen die Eigenschaft, dass in geklammerten Ausdrücken — *e.g.* in arithmetischen Ausdrücken — bei der Verschachtelung von Klammern zu jeder öffnenden auch eine dazu passende schließende Klammer vorhanden sein muss. Dies ist zum Beispiel der Fall für einen arithmetischen Ausdruck wie

$$((x + (3 \cdot y)) \cdot 2).$$

Dies ist eine Mindestanforderung an die korrekte Verwendung von Klammern. Aus dem Satz [3.14] folgt, dass zur Beschreibung dieser Eigenschaft reguläre Sprachen nicht geeignet sind, sondern dass dafür kontextfreie Sprachen benötigt werden. Deshalb ist es eine zentrale Aufgabe des Parsers und nicht des Scanners, die korrekte Klammerung zu prüfen.

[12] Siehe beispielsweise [40], Chapter 2, Theorem 2.29

3.11 Reguläre Ausdrücke

Wir haben bereits in Kapitel 2 gesehen, dass sich generell zur Beschreibung kontextfreier Sprachen in Informatikanwendungen die Erweiterte Backus–Naur Form besser eignet als die Theorie–basierte Schreibweise von Grammatiken. Der Hauptgrund liegt in einer besseren maschinellen Lesbarkeit der Backus–Naur Form.

Für die effiziente Verarbeitung regulärer Sprachen hat sich ein für diesen Sprachtyp speziell entwickelter Formalismus als noch geeigneter erweisen. Dies sind die **Regulären Ausdrücke**.

Reguläre Ausdrücke werden beispielweise bei der Suche nach Wortmustern in Editoren verwendet, bei der Filterung von Mails[13] oder als Spezifikationssprache für Scanner–Generatoren. Moderne Programmiersprachen wie Java bieten Bibliotheken für den Einsatz regulärer Ausdrücke an. In zahlreichen Anwendungen wird nur ein Teil der regulären Ausdrücke verwendet, beispielsweise in Form von Wildcarts bei der Suche nach Dateien un DOS oder UNIX.[14]

Reguläre Ausdrücke werden mit Hilfe von Operatoren für die Bildung von Alternativen, Sequenzen und Wiederholungen formuliert. Diese basieren berughen auf den mengentheoretischen Operationen Vereinigung, Konkatenation und Kleene Abschluß.

Man legt zunächst die Grundsymbole für reguläre Ausdrücke fest und beschreibt dann, wie man zusammengesetzte reguläre Ausdrücke bilden kann. Eine solche Definition nennt man *induktiv über den Aufbau* der Ausdrücke. Dies ist ein allgemeines Prinzip, das in der Informatik häufig verwendet wird.

[13] um beispielsweise Spam zu unterdrücken

[14] Eine umfassende Darstellung und Untersuchung von regulären Ausdrücken sowie deren praktische Verwendung findet man in dem Buch von Jeffrey Friedl [62].

Definition [3.18]:

Sei Σ ein Alphabet.

(a) $\emptyset$ ist ein regulärer Ausdruck für die Sprache $L(\emptyset) = \emptyset$.

(b) λ ist ein regulärer Ausdruck für die Sprache $L(\lambda) = \{\lambda\}$.

(c) Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck für die Sprache $L(a) = \{a\}$.

(d) Seien α und β reguläre Ausdrücke für die Sprachen $L(\alpha)$ bzw. $L(\beta)$. Dann sind auch folgende Ausdrücke *reguläre Ausdrücke*:

$\alpha \mid \beta$ Alternative
Es gilt

$$L(\alpha \mid \beta) = L(\alpha) \cup L(\beta) = \{w \mid w \in L(\alpha) \vee L(\beta)\}.$$

$\alpha\beta$ Sequenz
Es ist

$$\begin{aligned} L(\alpha\beta) &= L(\alpha)L(\beta) \\ &= L(\alpha) \circ L(\beta) = \{wv \mid w \in L(\alpha) \text{ und } v \in L(\beta)\}. \end{aligned}$$

Diese Operation heißt **Konkatenation**.

α^* Wiederholung
Es gilt:

$$\begin{aligned} L(\alpha^*) &= (L(\alpha))^* \\ &= \{\lambda\} \cup L(\alpha) \cup (L(\alpha))^2 \cup \cdots \\ &= \{w_1 w_2 \ldots w_n \mid w_i \in L(\alpha), i = 1, 2, \ldots, n, n \in \mathbb{N}\}. \end{aligned}$$

Der Operator * heißt **Verkettungshülle** oder **Kleene–Stern**.

Anmerkungen:

- Zur Betonung des zugrundeliegenden Alphabets verwendet man auch die Sprachregelung *regulärer Ausdruck über* Σ.
- Die Symbole $\emptyset, \lambda$ und die Elemente $a \in \Sigma$ werden hier mehrdeutig verwendet. Beispielsweise bezeichnet a in der Regel 3 ein Zeichen aus Σ, ein regulärer Ausdruck, sowie ein Wort über Σ, bestehend aus einem Zeichen. Da aus der Verwendung dieser Symbole normalerweise ersichtlich ist, welche Bedeutung sie heban, verzichtet man in der Regel auf unterschiedliche

Notationen.

- Für die Kennzeichnung der Konkatenationsoperation wird gelegentlich das Verknüpfungszeichen $\circ$ verwendet. Üblich ist, dass bei der Konkatention zweier Strings x und y, die Strings einfach (ohne Verknüpfungszeichen) hintereinandergesetzt werden.

Zur Strukturierung von regulären Ausdrücken werden runde Klammern benutzt. Um die Anzahl der Klammern auf das Nötigste zu reduzieren, legt man die folgenden Regeln zugrunde:

-¿ Der KLEENE–Stern bindet stärker als die Konkatenation.

-¿ Die Konkatenation bindet stärker als die Alternativenbildung.

Damit lässt sich beispielsweise die folgende Vereinfachung eines regulären Ausdrucks erhalten:

$$(a(b^*))^* \mid (ab) \longrightarrow (ab^*)^* \mid ab$$

Beispiel [3.51]

Wir listen in diesem Beispiel eine Reihe regulärer Ausdrücke auf, zusammen mit den zugehörigen Sprachen.

Regulärer Ausdruck α	Zugehörige Sprache $L(\alpha)$
$a \mid b$	$\{a, b\}$
$(a \mid b)(a \mid b)$	$\{aa, ab, ba, bb\}$
$a(a \mid b) \mid b(a \mid b)$	$\{aa, ab, ba, bb\}$
a^*	$\{a^n, n \geq 0\} = \{\lambda, a, aa, aaa, \cdots\}$
$\emptyset^*$	$\{\lambda\}$
λ^*	$\{\lambda\}$
ab^*	$\{ab^n \mid n \geq 0\}$
a^*b^*	$\{a^n b^m \mid n \geq 0, m \geq 0\}$
$(ab)^*$	$\{(ab)^n \mid n \geq 0\}$
$(a \mid b)^*$	$\{a, b\}^*$
$a^* \mid b^*$	$\{a^n \mid n \geq 0\} \cup \{b^m \mid m \geq 0\}$
$(ab^*)^*$	$\{\lambda\} \cup \{aw \mid w \in \{a, b\}^*\}$
$(aa \mid b)^*$	$\{w \in \{a, b\}^* \mid a$ kommt in w nur in Blöcken gerader Länge vor$\}$
$(\mathtt{0} \mid \mathtt{1}^*\mathtt{0}(\mathtt{0} \mid \mathtt{1})$	$\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid w$ hat an vorletzter Stelle eine $\mathtt{0}\}$.

Das zweite und das dritte Beispiel machen deutlich, dass die Darstellung einer Sprache durch einen regulären Ausdruck nicht eindeutig ist.

Rechenregeln für reguläre Ausdrücke

Wir bezeichnen mit $\mathcal{A}$ die Menge der regulären Ausdrücke. Auf der Menge $\mathcal{A}$ sind zwei binäre Operationen definiert, die Alternative sowie die Sequenz:

$$\begin{aligned} | : \mathcal{A} \times \mathcal{A} &\longrightarrow \mathcal{A} \\ (\alpha, \beta) &\longmapsto \alpha \mid \beta \end{aligned}$$

und

$$\begin{aligned} \circ : \mathcal{A} \times \mathcal{A} &\longrightarrow \mathcal{A} \\ (\alpha, \beta) &\longmapsto \alpha \circ \beta = \alpha\beta \end{aligned}$$

und eine unäre Abbildung (Kleene–Stern):

$$\begin{aligned} * : \mathcal{A} &\longrightarrow \mathcal{A} \\ \alpha &\longmapsto \alpha^*. \end{aligned}$$

In der folgenden Zusammenstellung sind Rechenregeln aufgelistet, mit deren Hilfe reguläre Ausdrücke vereinfacht werden können. Die verwendete Notation $\alpha \equiv \beta$ hat die Bedeutung, dass zwei reguläre Ausdrücke *alpha* und β äquivalent sind, wenn sie die gleiche reguläre Sprache beschreiben, *i.e.*

$$\alpha \equiv \beta \quad \Longleftrightarrow \quad L(\alpha) = L(\beta).$$

Für $\alpha, \beta, \gamma \in \mathcal{A}$ gilt

A1 Assoziativgesetze

$$\begin{aligned} (\alpha \mid \beta) \mid \gamma &\equiv \alpha \mid (\beta \mid \gamma), \\ (\alpha \circ \beta) \circ \gamma &\equiv \alpha \circ (\beta \circ \gamma). \end{aligned}$$

A2 Kommutativgesetz

$$\alpha \mid \beta \equiv \beta \mid \alpha.$$

A3 Distributivgesetze

$$\begin{aligned} \alpha \circ (\beta \mid \gamma) &\equiv \alpha \circ \beta \mid \alpha \circ \gamma, \\ (\alpha \mid \beta) \circ \gamma &\equiv \alpha \circ \gamma \mid \beta \circ \gamma. \end{aligned}$$

A4 Neutrale Elemente

Für die Alternative und Sequenz existeiern neutrale Element, *i.e.*

$$\begin{aligned} \alpha \mid \emptyset &\equiv \emptyset \mid \alpha \equiv \alpha, \\ \alpha \circ \lambda &\equiv \lambda \circ \alpha \equiv \alpha. \end{aligned}$$

A5 Annihilation

$$\alpha \circ \emptyset \equiv \emptyset \circ \alpha \equiv \emptyset.$$

Die Axiome A1 bis A5 definieren einen Semiring. Wir benötigen noch eine weiter Regel

A6 Idempotenz

$$\alpha \mid \alpha \equiv \alpha$$
$$(\alpha^*)^* \equiv \alpha^*.$$

Das 6–Tupel

$$\mathcal{K} = (\mathcal{A}, |, \circ, *, \emptyset, \lambda)$$

mit den Axiomen A1 bis A6 ist die algenraische Struktur eine **Kleene Algebra**.

Anmerkung:

Für die Alternativbildung | verwendet man häufig auch das +–Zeichen, megentheoretisch ist die die Vereinigung $\cup$.

3.11.1 Reguläre Ausdrücke und reguläre Sprachen

Die regulären Ausdrücke bilden neben regulären Grammatiken, DFA und NFA einen weiteren Formalismus zur Beschreibung regulärer Sprachen. Die einfache Syntax regulärer Ausdrücke erlaubt eine effiziente Verarbeitung in Rechnern. Dass damit alle regulären Sprachen erfasst werden ist Thema dieses Abschnittes.

Theorem [3.16]:

> Zu jedem regulären Ausdruck gibt es einen `NFA`, der die durch den regulären Ausdruck generierte Sprache akzeptiert.

Beweis:

Zu zeigen ist: Gegeben ist ein beliebiger regulärer Ausdruck. Dann konstruieren wir eine NFA aus diesem regulären Ausdruck.

Der Beweis wird induktiv über den Aufbau der regulären Ausdrücke ausgeführt. Dadurch ergibt sich ein systematisches Verfahren, mit dem man Schritt für Schritt aus den endlichen Automaten für die Grundsymbole endliche Automaten für beliebig zusammengesetzte reguläre Ausdrücke konstruiert. Wir stellen den Konstruktionsalgorithmus vor, unterlegt jeweils mit einem Beispiel.

Sei Σ ein Alphabet. Für den Induktionsanfang geben wir die folgenden drei NFA an, die jeweils die Sprache, die von den regulären Ausdrücken

$$\alpha = \emptyset, \alpha = \lambda \text{ und } \alpha = a,\ a \in \Sigma$$

dargestellt wird, erkennen.

$\alpha = \emptyset$ $\alpha = \lambda$ $\alpha = a$

Die Aussage gelte nun für die beiden regulären Ausdrücke α_1 und α_2. Wir zeigen nun die Behauptung für die drei regulären Ausdrücke $\alpha = \alpha_1 \mid \alpha_2, \alpha = \alpha_1 \circ \alpha_2$ und $\alpha = \alpha_1^*$.

$\alpha = \alpha_1 \mid \alpha_2$ **Vereinigung zweier nichtdeterministischen endlichen Automaten**.
Gegeben sind die beiden regulären Ausdrücke α_1 und α_2, sowie die beiden NFA

$$N_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$

mit $L(N_1) = L(\alpha_1)$ und

$$N_2 = (Q_2, \Sigma, \delta_2, q_2^s, F_2)$$

mit $L(N_2) = L(\alpha_2)$. O.B.d.A gelte $Q_1 \cap Q_2 = \emptyset$.

Dann ist

$$N = (Q, \Sigma, \delta, q^s, F)$$

mit

$$Q = Q_1 \cup Q_2 \cup \{q^s\}, \quad q^s \notin Q_1 \cup Q_2,$$

q^s ist ein neuer Startzustand. Die Übergangsrelation ist

$$\begin{aligned} \delta = \delta_1 \cup \delta_2 \\ \cup \{(q^s, a, q) \mid (q_1^s, a, q) \in \delta_1\} \\ \cup \{(q^s, a, q) \mid (q_2^s, a, q) \in \delta_2\} \end{aligned}$$

i.e. der Produktautomat hat die gleichen Übergänge wie die beiden Teilautomaten, hinzufügen müssen wir neue Übergänge von dem neuen Startzustand q^s, diese sind die gleichen Übergänge wie die Teilautomaten von ihren Startzuständen haben. Die Menge der Acceptzustände ist:

$$F = \begin{cases} F_1 \cup F_2 \cup \{q^s\} & \text{falls } q_1^s \in F_1 \vee q_2^s \in F_2 \\ F_1 \cup F_2 & \text{sonst.} \end{cases}$$

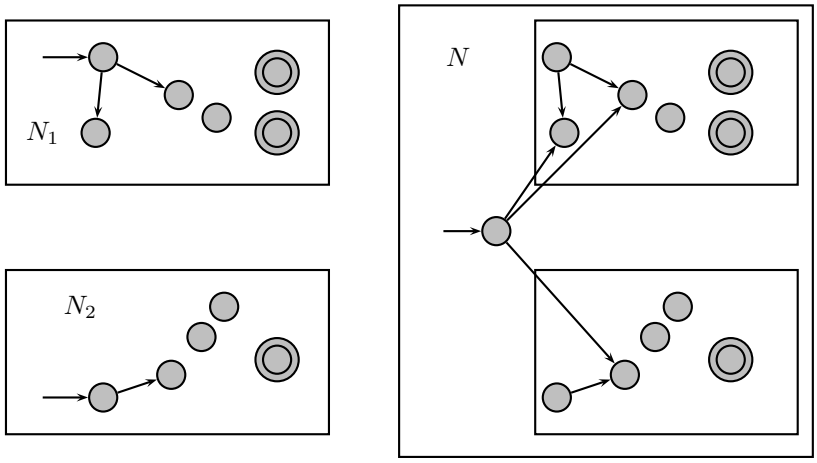
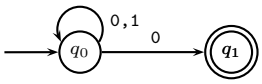


Abbildung 3.28: Konstruktion eines NFA N, der die Sprache $L_1 \cup L_2$ akzeptiert.

Beispiel [3.52]

Sei $\alpha_1 = (\mathtt{0} \mid \mathtt{1})^*\mathtt{0}$. Dieser reguläre Ausdruck erzeugt alle Bitstrings, die mit 0 enden. Ein NFA, der diese Sprache akzeptiert ist



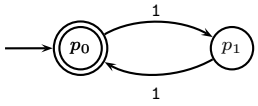
Formal haben wir

$$N_1 = (\{q_0, q_1\}, \{\mathtt{0}, \mathtt{1}\}, \delta_1, q_0, \{q_1\})$$

mit

	0	1
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\emptyset$

Sei $\alpha_2 = (\mathtt{11})^*$, *i.e.* dieser reguläre Ausdruck erzeugt alle Bitstrings, die eine gerade Anzahl von 1en enthalten. Ein NFA, der diese Sprache akzeptiert ist:



Formal ist:

$$N_2 = (\{p_0, p_1\}, \{\mathtt{0}, \mathtt{1}\}, \delta_2, p_0, \{p_1\})$$

mit der Übergangsrelation δ_2:

	0	1
p_0	$\emptyset$	$\{p_1\}$
p_1	$\emptyset$	$\{p_0\}$

Dann liefert die Konstruktion den Automaten

$$N = (Q, \{\mathtt{0}, \mathtt{1}\}, \delta, q^s, F)$$

mit

$$Q = Q_1 \cup Q_2 \cup \{q_{neu}\} = \{q_{neu}, q_0, q_1, p_0, p_1\},$$

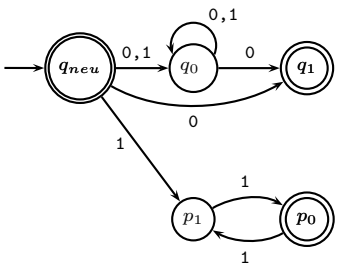
dem neuen Startzustand $q^s = q_{neu}$, der Menge der Acceptzustände

$$F = \begin{cases} F_1 \cup F_2 \cup \{q^s\} & \text{falls } q_1^s \in F_1 \vee q_2^s \in F_2 \\ F_1 \cup F_2 & \text{sonst.} \end{cases}$$
$$= \{q_1, p_0, q_{neu}\}.$$

Die Übergangsrelation ist:

	0	1
q_{neu}	$\{q_0, q_1\}$	$\{q_0, p_1\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\emptyset$
p_0	$\emptyset$	$\{p_1\}$
p_1	$\emptyset$	$\{p_0\}$

Das Zustandsdiagramm ist



$\alpha = \alpha_1 \circ \alpha_2$ **Konkatenation zweier NFA**

Gegeben sind die beiden regulären Ausdrücke α_1 und α_2 und die beiden `NFA`

$$N_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$

mit $L(N_1) = L(\alpha_1)$ und

$$N_2 = (Q_2, \Sigma, \delta_2, q_2^s, F_2)$$

mit $L(N_2) = L(\alpha_2)$, o.B.d.A gelte $Q_1 \cap Q_2 = \emptyset$.

Die Beweisidee ist die folgende: Wir kennen einen `NFA` N_1, der die Sprache $L(\alpha_1)$ akzeptiert, sowie einen `NFA` N_2, der die Sprache $L(\alpha_2)$ akzeptiert. Um einen endlichen Automaten (Produktautomaten) N zu erhalten, der Worte der Form $w_1 w_2$ mit $w_1 \in L(\alpha_1), w_2 \in L(\alpha_2)$ akzeptiert, hilft die folgende Überlegung: Der Produktautomat muss zunächst das Teilwort w_1 verarbeiten, dazu arbeitet er genau wir der Automat N_1. Hat er das Teilwort w_1 komplett verarbeitet, dann befindet sich der Produktautomat in einem der Acceptzustände des Teilautomaten N_1. Er ist von einem Zustand $q \in Q_1$ beim Verarbeiten des Zeichens $a \in \Sigma$ in einen der Acceptzustände $p \in F_1$ übergegangen.

Der Produktautomat muss nun mit der Verarbeitung des Teilworts w_2 fortfahren. Dazu verfährt er wieder exakt wie der Teilautomat N_2. Er muss dazu aber von der Zuständen des Teilautomaten N_1 in den Startzustand des Automaten N_2 gelangen, und zwar so, dass dann und nur dann das Teilwort w_2 verarbeitet wird, wenn dass w_1 Teilwort erfolgreich (von N_1) verarbeitet wurde. Wir können die geeignete Ankopplung des Automaten N_2 an N_1 dadurch erhalten, dass alle Übergänge im Automaten N_1, die in die Acceptzutände übergehen, erweitert werden und in den Startzustand des Automaten N_2 gehen.

Wir konstruieren den NFA, der die Sprache $L(\alpha_1) \circ L(\alpha_2)$ akzeptiert, wie folgt:

$$N = (Q, \Sigma, \delta, q^s, F)$$

mit

$$Q = Q_1 \cup Q_2,$$

dem Startszustand q^s. Für die Übergangsrelation unterscheidet man drei Fälle:

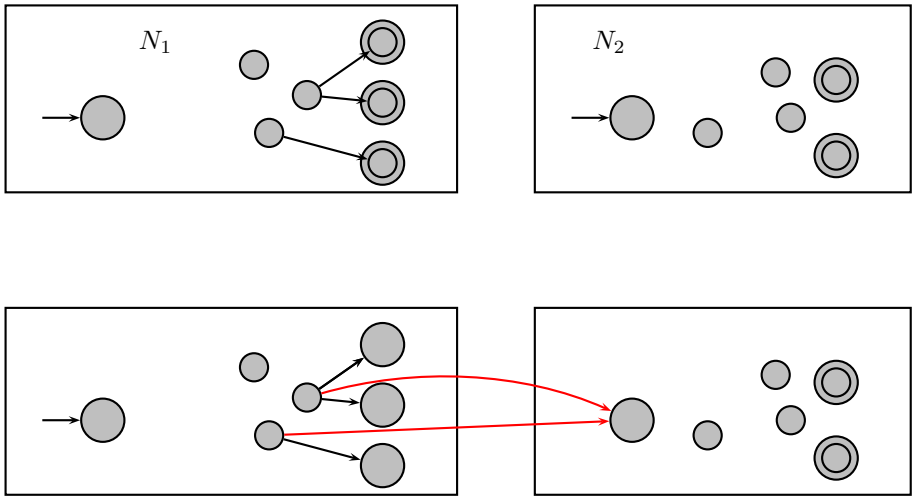


Abbildung 3.29: Konstruktion des Produktautomaten.

Fall 1: $q_1^s \notin F_1$:

$$\begin{aligned}\delta &= \delta_1 \cup \delta_2 \\ &\quad \cup \{(p, a, q_2^s) \mid (p, a, q) \in \delta_1, q \in F_1\}. \\ F &= F_2.\end{aligned}$$

Dieser Übergangsrelation liegt die folgende Idee zugrunde. Der Produktautomat muss Worte der Form $x = w_1 w_2$ akzeptieren, wobei der NFA N_1 das Teilwort w_1 akzeptiert und N_2 den Postfix w_2. Der zusammengesetzte Automat simuliert zunächst den Automaten N_1 bei der Verarbeitung von w_1 und muss in einem der Acceptzustände sein, wenn der Präfix w_1 verarbeitet ist.

Fall 2: $q_1^s \in F_1, q_2^s \notin F_2$:
Erweitere die Übergangsrelation δ aus Fall 1 um

$$\{(q_1^s, a, q) \mid (q_2^s, a, q) \in \delta_2\},$$

$$F = F_2.$$

Fall 3: $q_1^s \in F_1, q_2^s \in F_2$:
Wähle δ wie in Fall 2 und setze

$$F = F_2 \cup \{q_1^s\}.$$

Beispiel [3.53]

Seien die folgenden beiden regulären Ausdrücke gegeben:

$$\alpha_1 = \mathtt{1}^*\mathtt{0}(\mathtt{0} \mid \mathtt{1})^*, \qquad \alpha_2 = (\mathtt{11})^*.$$

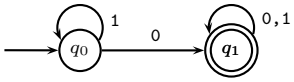
Ein NFA, der die von α_1 erzeugte Sprache akzeptiert ist formal:

$$N_1 = (\{q_0, q_1\}, \{\mathtt{0}, \mathtt{1}\}, \delta_1, q_0, \{q_1\})$$

mit

	0	1
q_0	$\{q_1\}$	$\{q_0\}$
q_1	$\{q_1\}$	$\{q_1\}$

Das Zustandsdiagramm ist:



Der Automat N_2 mit $L(N_2) = L(\alpha_2)$ ist identisch mit dem NFA N_2 aus dem vorigen Beispiel [3.52].

Dann liefert die Konstruktion den Automaten, der die Sprache $L(\alpha_1 \circ \alpha_2)$ akzeptiert, folgendermaßen:

$$N = (Q, \{\mathtt{0}, \mathtt{1}\}, \delta, q^s, F)$$

mit der Menge von Zuständen

$$Q = Q_1 \cup Q_2 = \{q_0, q_1, p_0, p_1\},$$

und dem Startzustand $q^s = q_1^s = q_0$. In diesem Beispiel haben wir Fall 1 vorliegen, *i.e.* der Startzustand des NFA N_1 ist kein Acceptzustand. Daher ist

$$\begin{aligned} \delta &= \delta_1 \cup \delta_2 \\ &\quad \cup \{(p, a, q_2^s) \mid (p, a, q) \in \delta_1, q \in F_1\}. \\ F &= F_2. \end{aligned}$$

Das bedeutet, die Übergangsrelation des NFA N enthält zunächst die Übergänge der beiden einzelnen Automaten N_1 und N_2. Zusätzlich werden Übergänge hinzugefügt, die von allen Acceptzuständen des Automaten N_1 — diese sind in N keine Acceptzustände mehr — in den Startzustand des N_2–Automaten übergehen. Diese zusätzlichen Übergänge sind identisch mit den Übergängen

$$\delta_1(q, a) = p, \quad p \in F_1.$$

Der Acceptzustand des Automaten N_1 ist q_1. In diesen Acceptzustand haben wir die folgenden drei Übergänge, die *in* q_1 landen.

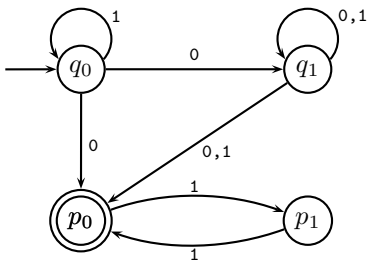
$$\begin{aligned}\delta_1(q_0, \mathtt{0}) &= q_1,\\ \delta_1(q_1, \mathtt{0}) &= q_1,\\ \delta_1(q_1, \mathtt{1}) &= q_1.\end{aligned}$$

Daher haben wir drei zusätzliche Übergänge in den Startzustand des Automaten N_2, i.e. p_0:

$$\begin{aligned}\delta(q_0, \mathtt{0}) &= p_0,\\ \delta(q_1, \mathtt{0}) &= p_0,\\ \delta(q_1, \mathtt{1}) &= p_0.\end{aligned}$$

	0	1
q_0	$\{q_1, p_0\}$	$\{q_0\}$
q_1	$\{q_1, p_0\}$	$\{q_1, p_0\}$
p_0	$\emptyset$	$\{p_1\}$
p_1	$\emptyset$	$\{p_0\}$.

Das Zustandsdiagramm dieses Automaten ist



$\alpha = \alpha_1^*$ **Mehrfachausführung, Verkettungshülle eines nichtdeterministischen endlichen Automaten**
Gegeben ist der reguläre Ausdruck α_1 mit der dadurch generierten Sprache $L(\alpha_1)$. Weiterhin gibt es einen NFA

$$N_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$

mit $L(N_1) = L(\alpha_1)$.

Dann gibt es einen NFA

$$N = (Q, \Sigma, \delta, q^s, F)$$

mit $L(N) = L(\alpha_1)$,definiert wie folgt:

Fall 1: $q_1^s \in F_1$, dann ist

$$Q = Q_1, q^s = q_1^s, F = F_1$$

und die Übergangsrelation ist:

$$\delta = \delta_1 \cup \{(p, a, q) \mid (p, a, q) \in \delta_1, q \in F_1\}$$

Fall 2: $q_1^s \notin F_1$. Wie wählen einen neuen Startzustand $q_n \notin Q_1$ mit $q^s = q_n$

$$Q = Q_1 \cup \{q_n\}, q^s = q_{neu}, F = F_1 \cup \{q_{neu}\},$$

und

$$\begin{aligned}\delta = \delta_1 & \\ & \cup \{(p, a, q_n) \mid (p, a, q) \in \delta_1, q \in F_1\} \\ & \cup \{(q_n, a, q) \mid (q_1^s, a, q) \in \delta_1\} \\ & \cup \{(q_n, a, q_n) \mid (q_1^s, a, q) \in \delta_1, q \in F_1\}\end{aligned}$$

Die Beweisidee in diesem Fall ist wie folgt. i

Es ist zunächst zweckmäßig, sich nochmals die Definition der KLEENE Star Operation anzusehen:

$$L^* = \{w_1 w_2 \dots w_k \mid k \geq 0 \text{ und } w_i \in L \text{ für alle } i, 0 \leq i \leq k\},$$

mit anderen Worten: L^* enthält alle möglichen Konkatenationen, die sich aus Worten der Sprache L bilden lassen.

Wir betrachten einen nichtdeterministischen endlichen Automaten N_1, der die Sprache L erkennt. Dieser Automat wird modifiziert, sodass er die Sprache L^* erkennt. Dies ist in der Abbildung [3.30] dargestellt. Der resultierende `NFA` N akzeptiert den Input genau dann, wenn dieser in mehrere

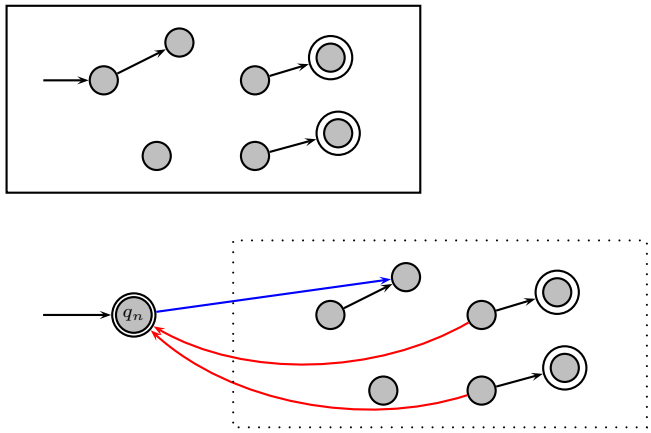


Abbildung 3.30: Konstruktion des nichtdeterministischen endlichen Automaten N, der die Sprache L^* erkennt.

Teilstrings zerlegt werden kann, wobei jeder Teilstring von N_1 akzeptiert wird.

Prinzipiell können wir den Automaten N genauso aufbauen wie den Automaten N_1. Zunächst ist zu bemerken, dass L^* das leere Wort λ enthält, daher müssen wir einen neuen Startzustand einbauen q_n, der auch Acceptzustand ist. Zusätzlich benötigen wir Übergänge von allen Zuständen, die in die Acceptzustände des Automaten N_1 übergehen, in den neuen Startzustand. Dadurch hat die Maschine N die Möglichkeit, bei der Verarbeitung eines Strings in den Startzustand zurückzuspringen und einen weiteren Teilstring zu verarbeiten, der von N_1 akzeptiert wird. Dies sind die Übergänge

$$\{(p, a, q_n) \mid (p, a, q) \in \delta_1 \wedge q \in F_1\}.$$

Weiterhin muss der Automat N von dem neuen Startzustand q_n in die gleichen Zustände übergehen wie der Automat N_1 von dem ursprünglichen Startzustand. Dies sind die Übergänge

$$\{(q_n, a, q) \mid (q_1^s, a, q) \in \delta_1\}$$

Wir müssen noch weitere Übergänge hinzufügen. Direkte Übergänge vom alten Startzustand q_1^s in einen Acceptzustand werden von dem neuen Automaten N nicht mehr ausgeführt, da es keinen direkten Übergang vom neuen Startzustand q_n in den alten Startzustand q_1^2 gibt. Unter Umständen ist der alte Startzustand nicht mehr erreichbar. Daher fügen wir Übergänge hinzu, die den neuen Startzustand in sich überführt. Das sind die Übergänge der Form

$$\{(q_n, a, q_n) \mid (q_1^s, a, q) \in \delta_1 \wedge q \in F_1\}.$$

Beispiel [3.54]

Betrachte den regulären Ausdruck $\alpha_1 = \mathtt{0}(\mathtt{00})^*(\lambda \mid \mathtt{1})$. Dieser erzeugt die Sprache der Bitstrings, die eine ungerade Anzahl von `0` enthalten und mit `1` enden, oder keine `1` am Ende haben,

$$L = \{\mathtt{0}, \mathtt{01}, \mathtt{000}, \mathtt{0001}, \mathtt{00000}, \mathtt{000001}, \ldots\}$$

Ein NFA N_1, der diese Sprache akzeptiert ist formal:

$$N_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$

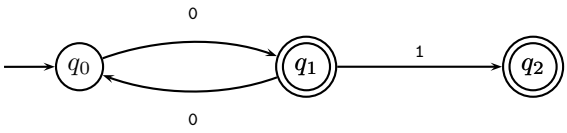
mit

$$\begin{aligned} Q_1 &= \{q_0, q_1, q_2\}, \\ \Sigma &= \{\mathtt{0}, \mathtt{1}\}, \\ q_1^s &= q_0, \\ F_1 &= \{q_1, q_2\}, \end{aligned}$$

und der Übergangsrelation δ_1:

	`0`	`1`
q_0	$\{q_1\}$	$\emptyset$
q_1	$\{q_0\}$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

Das Zustandsdiagramm ist



Den Automaten N, der den Kleene–Abschluß der Sprache $L(\alpha_1)$ erkennt, *i.e.* die von dem regulären Ausdruck

$$\alpha = (\mathtt{0}(\mathtt{00})^*(\lambda \mid \mathtt{1}))^*$$

erzeugte Sprache, konstruieren wir folgendermaßen: Da der Startzustand q_0 des Automaten N_1 kein Acceptzustand ist, wählen wir einen neuen

Startzustand q_{neu}. Die Menge der Zustände des NFA N besteht aus den vier Zuständen

$$Q = \{q_{neu}, q_0, q_1, q_2\}.$$

Der Startzustand ist q_{neu} und die Menge der Acceptzustände ist

$$F = F_1 \cup \{q_{neu}\} = \{q_{neu}, q_1, q_2\}.$$

Zur Konstruktion der Übergangsrelation sehen wir uns die zusätzlichen Übergänge im Einzelnen an:

$$\begin{aligned}\delta = \delta_1 & \\ & \cup \{(p, a, q_{neu}) \mid (p, a, q) \in \delta_1, q \in F_1\} \\ & \cup \{(q_{neu}, a, q) \mid (q_1^s, a, q) \in \delta_1\} \\ & \cup \{(q_{neu}, a, q_{neu}) \mid (q_1^s, a, q) \in \delta_1, q \in F_1\}\end{aligned}$$

(a) $\{(p, a, q_{neu}) \mid (p, a, q) \in \delta_1, q \in F_1\}$
Es ist

$$(p, a, q) \in \delta_1, q \in F_1 \Longleftrightarrow \delta_1(p, a) = q; q \in F_1.$$

Daher hat die Menge $\{(p, a, q_{neu}) \mid (p, a, q) \in \delta_1, q \in F_1\}$ die Bedeutung: Für jeden Übergang der Maschine, der in einen Acceptzustand übergeht, füge einen Übergang in den neuen Startzustand hinzu. Dies sind für die Maschine N_1 die beiden Übergänge

$$\begin{aligned}\delta_1(q_0, \mathtt{0}) &= q_1, \\ \delta_1(q_1, \mathtt{1}) &= q_2.\end{aligned}$$

Daher hat die Maschine N die beiden Übergänge

$$\begin{aligned}\delta(q_0, \mathtt{0}) &= q_{neu}, \\ \delta(q_1, \mathtt{1}) &= q_{neu}.\end{aligned}$$

(b) $\{(q_{neu}, a, q) \mid (q_1^s, a, q) \in \delta_1\}$
In diesem Fall haben wir

$$(q_1^s, a, q) \in \delta_1 \Longleftrightarrow \delta_1(q_0, a) = q.$$

Damit hat $\{(q_{neu}, a, q) \mid (q_1^s, a, q) \in \delta_1\}$ die Bedeutung, dass für jeden Übergang im Automaten N_1 vom Startzustand q_0 zu einem anderen Zustand q des Automaten N_1 in der Maschine N ein Übergang vom neuen Startzustand in den entsprechenden Zustand q gibt. Für N_1 ist dies der Übergang

$$\delta(q_0, \mathtt{0}) = \{q_1\}.$$

Daher ist der Maschine N der Übergang

$$\delta(q_{neu}, \mathtt{0}) = \{q_1\}$$

hinzuzufügen.

(c) $\{(q_{neu}, a, q_{neu}) \mid (q_1^s, a, q) \in \delta_1, q \in F_1\}$
Schließlich haben wir

$$(q_1^s, a, q) \in \delta_1, q \in F_1 \iff \delta_1(q_0, a) = q, q \in F_1.$$

Das heisst, für alle Übergänge im Automaten N_1, die vom (alten) Startzustand direkt in einen Acceptzustand übergehen, füge im Automaten N einen Übergang hinzu, der den neuen Startzustand q_{neu} in sich überführt. Im vorliegenden Fall ist

$$\delta_1(q_0, \mathtt{0}) = q_1, \quad q_1 \in F_1.$$

Daher fügen wir den Übergang

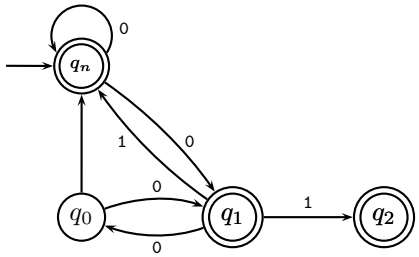
$$\delta(q_{neu}, \mathtt{0}) = q_{neu}$$

hinzu.

Dadurch ergibt sich die folgende Übergangsrelation.

	0	1
q_{neu}	$\{q_{neu}, q_1\}$	$\emptyset$
q_0	$\{q_{neu}, q_1\}$	$\emptyset$
q_1	$\{q_0\}$	$\{q_{neu}, q_2\}$
q_2	$\emptyset$	$\emptyset$.

Das Zustandsdiagramm dieses NFA ist (mit $q_n = q_{neu}$):



Anmerkung:

Dieser Satz, den wir soeben bewiesen haben, sagt insbesondere aus, dass die Klasse der reguläre Sprachen abgechlossen ist unter Vereinigung, Konkatenation und KLEENE Stern. Denn: Wir haben gezeigt, wenn

- L_1 und L_2 regulär sind, dann gibt es `NFA` N_1 und N_2 mit $L_i = L(N_i), i = 1, 2$. Dann gibt es auch einen `NFA` N mit $L_1 \cup L_2 = L(N) \Longrightarrow L_1 \cup L_2$ ist regulär.

- L_1 und L_2 regulär sind, dann gibt es `NFA` N_1 und N_2 mit $L_i = L(N_i), i = 1, 2$. Dann gibt es auch einen `NFA` N mit $L_1 \circ L_2 = L(N) \Longrightarrow L_1 \circ L_2$ ist regulär.

- L_1 regulär ist, dann gibt es einen `NFA` N_1 mit $L_1 = L(N_1)$. Dann gibt es auch einen `NFA` N mit $(L_1)^* = L(N) \Longrightarrow (L_1)^*$ ist regulär.

Man kann weiter zeigen, dass die Klasse der regulären Sprachen auch abgeschlossen ist unter Durchschnitt und Komplement.

$\Longrightarrow$ Übung [3.35]

Theorem [3.17]:

Zu jeder regulären Sprache L gibt es einen regulären Ausdruck, der diese Sprache generiert.

Beweis:

Wir zeigen: Gegeben ist ein nichtdeterministischer endlicher Automat N, der die Sprache $L(N)$ akzeptiert. Für jeden NFA gibt es einen regulären Ausdruck α mit $L(N) = L(\alpha)$.

Zum Beweis wird ein konstruktives Verfahren angegeben, das einen beliebigen NFA in einen regulären Ausdruck umformt.[15] Als Illustration dieses Verfahrens dient der in der Abbildung [3.31] dargestellt NFA.

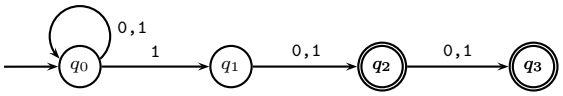


Abbildung 3.31: NFA, der in einen regulären Ausdruck umgewandelt wird.

[15] Die hier präsentierte Vorgehensweise ist etwas pragmatisch. Aus Zeitgründen verzichten wir hier auf eine mathematisch strenge Vorgehensweise. Diese findet man beispielsweise in dem Buch von Sipser [44].

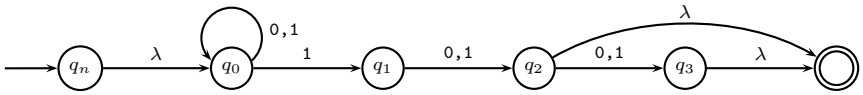


Abbildung 3.32: Erweiterter NFA

In dem Verfahren werden aus dem Zustandsdiagramm des NFA schrittweise die inneren Zustände entfernt und dafür die Kanten mit zunehmend komplexeren regulären Ausdrücken versehen. Das Verfahren terminiert, wenn der Zustandsgraph nur noch aus einem Start– und einem Endzustand und einem Zustandsübergang besteht. Die Beschriftung dieses Zustandsübergangs ist der gesuchte reguläre Ausdruck.

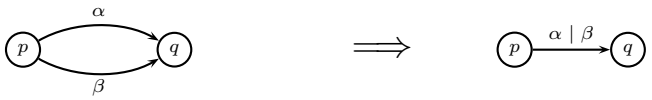
Zunächst wird ein neuer Startzustand eingeführt und durch einen λ–Übergang mit dem ursprünglichen Startzustand verbunden. Weiterhin wird ein neuer Endzustand eingeführt. Alle bisherigen Endzustände verlieren die Eigenschaft, Acceptzustand zu sein. Die ursprünglichen Endzustände werden ebenfalls mit λ–Übergänge mit dem neuen Endzustand verknüpft. Dies ist in der Abbildung [3.32] skizziert.

Außer dem Start– und Endzustand werden nun alle Zustände nach folgenden Regeln schrittweise eliminiert.

Wenn ein Zustand q_i entfernt wird, so wird jeder Kantenzug $(p, q_i)(q_i, q)$ mit $p, q \neq q_i$ durch eine Kante (p, q) ersetzt. Sind hierbei die Kanten (p, q_i) und (q_i, q) mit den regulären Ausdrücken α beziehungesweise β beschriftet, so wird die neue Kante (p, q)

- mit dem regulären Ausdruck $\alpha \circ \beta$ beschriftet, wenn keine Kante (q_i, q_i) vorhanden ist (siehe Abbildung [3.33] oben),
- mit dem regulären Ausdruck $\alpha \circ \gamma^* \circ \beta$ beschriftet, wenn ein Loop (q_i, q_i) vorhanden ist, der mit γ beschriftet ist (siehe Abbildung [3.33] unten).

Ist nach dem Entfernen eines Zustands eine Doppelkante zwischen zwei Zuständen vorhanden, so wird diese durch eine einfache Kante ersetzt. Die Beschriftung der Kante ist $\alpha \mid \beta$, wenn α bzw. β die Beschriftungen der Doppelkante war.



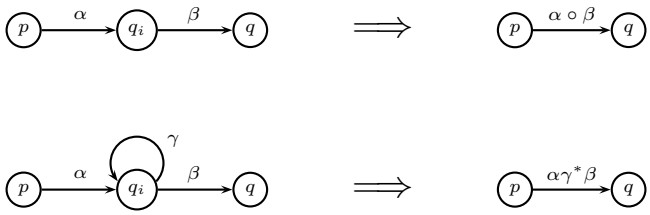
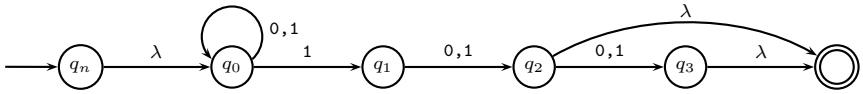


Abbildung 3.33: Elimination eines Zustandes. Ohne Loop (oben) und mit Loop (unten).

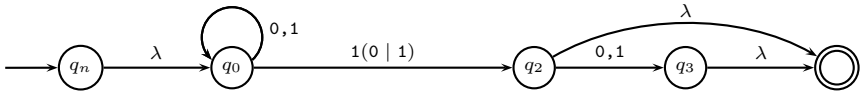
Beispiel [3.55]

Die folgenden Schritte zeigen die Anwendung dieses Verfahrens auf den Automaten aus der Abbildung [3.31].

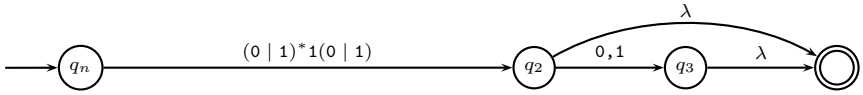
Erweiterung mit einem neuen Start– und Acceptzustand.



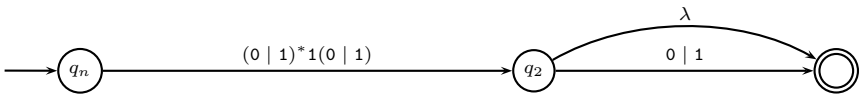
Elimination des Zustands q_1:



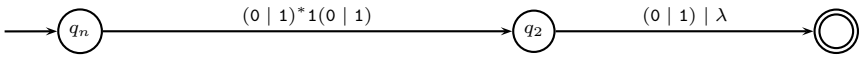
Elimination des Zustands q_0:



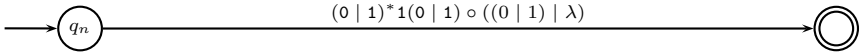
Elimination des Zustands q_3:



Ersetzung des doppelten Übergangs von q_2 in den Endzustand.



Elimination des Zustands q_2.



Damit haben wir den regulären Ausdruck abgeleitet:

$$\alpha = (0 \mid 1)^* 1 (0 \mid 1) \circ ((0 \mid 1) \mid \lambda)$$

⟹ Übungen [3.28] und [3.29].

3.12 Scanner

Der Scanner ist der Teil des Compilers, der die einzelnen Zeichen, die mit der Tastatur eingegeben wurden, zu **Token** zusammenfasst. Die Token bilden die zu analysierenden Eingabesymbole (*i.e.* die Elemente der Menge Σ) wie Schlüsselwörter, Bezeichner, Literale, Operatoren oder Begrenzer. Ist dies nicht möglich, dann ist die EIngabe kein syntaktisch korrektes Programm.

Jede Klasse von Token — also Schlüsselwörter, Bezeichner, etc. — kann als reguläre Sprache betrachtet werden. Daher kann für jede Tokenklasse ein endlicher Automat angelegt werden, der genau die Token dieser Klasse akzeptiert.

Endliche Automaten können leicht als Computerprogramm realisiert werden. Der aktuelle Zustand lässt sich in einer Variablen speichern. In eine Schleife wird pro Durchgang jeweils das nächste Zeichen des Inputstrings gelesen und mit Hilfe von Fallunterscheidungen in Abhängigkeit vom augenblicklichen Zustand und gelesenen Zeichen der Zustand neu gesetzt.

Wenn alle Token systematisch voneinander getrennt wären — beispielsweise durch *Whitespaces* wie Leerzeichen, Tabulator, Zeilenvorschub, Kommentar — so könnte jedes Token nacheinander von jedem dieser Automaten analysiert werden, bis es als Element einer Tokenklasse identifiziert ist. Falls das Token von keinem der endlichen Automaten akzeptiert wird, kann eine Fehlerbehandlung durchgeführt werden.

In Programmiersprachen ist oft das direkte Anfügen von Token erlaubt, in C ist ein Ausdruck der Form

$$\texttt{id2} + \texttt{1}$$

erlaubt, der aus den drei Token `id2` (Bezeichner), `+` (Operator), und `1` (Literal) besteht.

3.13 Abkürzungen für reguläre Ausdrücke

Aus Gründen der Maschinenlesbarkeit verwendet man eine Reihe von Abkürzungen und spezielle Notationen. Beispielsweise ist das Zeichen λ — das leere Wort — ale ASCII Zeichen nicht verfügbar.

Definition [3.19]:

Sei Σ ein Alphabet und α ein regulärer Ausdruck über Σ. In α dürfen die im Folgenden aufgeführten Abkürzungen enthalten sein. $L(\alpha)$ bezeichnet die Sprache, die durch *alpha* dargestellt ist.

α^+ Dieser Ausdruck bezeichnet die Sprache

$$L(\alpha)^+ = \{x_1 x_2 \dots x_n \mid x_i \in L(\alpha), i = 1, \dots, n, n \geq 1, n \in \mathbb{N}\},$$

das ein– oder mehrmalige Auftreten von α.

$\alpha?$ Dieser Ausdruck beschreibt die Sprache $L(\alpha) \cup \{\lambda\}$, *i.e.* der Ausdruck α tritt kein– oder einmal auf.

[Liste] Der Eintrag *Liste* hat die Form $a_1 a_2 \dots a_n$ oder $b-c$, wobei $a_1, a_2, \dots, a_n, b, c \in \Sigma$ ist und $b < c$ bezüglich einer auf Σ definierten linearen Ordnung ist. Dieser Ausdruck bezeichnet die Sprache

$$\{a_1, a_2, \dots, a_n\} \text{ bzw. } \{a \in \Sigma \mid b \leq a \leq c\}.$$

Dies ist eine Auflistung aller Zeichen oder eine Bereichsangabe.

. Der Punkt steht für ein beliebiges Zeichen aus Σ, er bezeichnet die Sprache

$$\{a \mid a \in \Sigma\}.$$

Beispiel [3.56]

Regulärer Ausdruck	Bedeutung (Sprache)
(D|d)(er|ie|as)	Groß oder klein geschriebener Artikel
M((e|a)(i|y)(e| λ)r	{ Meier, Meir, Meyer, Meyr, Maier, Mair,Mayer, Mayr }.
10^* Euro	{ 1 Euro, 10 Euro, 100 Euro, 1000 Euro,... }

Beispiel [3.57]

Der reguläre Ausdruck

$$\alpha = (+ \mid -)?[0-9]^+$$

bezeichnet die Sprache, die aus allen ganzen Zahlen (mit führenden Nullen) mit oder ohne Vorzeichen besteht.

Der reguläre Ausdruck

$$\alpha = [0-9A-F]^+$$

bezeichnet die Sprache, die aus allen Hexadezimalzahlen (mit führenden Nullen) besteht.

Für den praktischen Einsatz müssen für reguläre Ausdrücke weitere Konventionen verienbart sein, um einerseits die Eingabe auf der Tastatur zu ermöglichen, andererseits gewisse Mehrdeutigkeiten auszuschließen. Unter UNIX — beispielsweise mit dem Programm `grep` — gelten folgende Konventionen

- Die Zeichen $*$ und $+$ werden nicht hochgesetzt
- Es besteht mitunter die Notwendigkeit der *Maskierung* von Zeichen. Beispielsweise hat der Punkt die Bedeutung *ein beliebiges Zeichen aus* Σ. Wird aber der Punkt als Zeichen aus dem Alphabet verstanden, muss er maskiert werden. Dies geschieht mit einem vorangestellten Backslash, *i.e.* $\backslash$.

Beispiel [3.58]

Im Rahmen eines Softwareprojektes werden Eingabefelder für Kontostände erstellt. Neben den Eingabefeldern für Kontonummern und für Daten des Kontoinhabers ist ein Eingabefeld für Ein– und Auszahlungen vorgesehen.[16] Die Eingaben in diesem Feld müssen laut Spezifikation die folgende Gestalt haben.

> Zunächst ist die Eingabe der Währung erforderlich. Als Währungsangaben sind zulässig: \$ (US–Dollar), € (Euro) und £(engl. Pfund). Hinter der Währungsangabe folgt der Betrag. Der Betrag kann mit den Vorzeichen + bzw. − beginnen, muss aber kein Vorzeichen haben. Der Betrag besteht aus einem ganzzahligen Anteil und einem Dezimalanteil. Beide werden durch einen Punkt getrennt. Der ganzzahlige Anteil kann beliebig lang sein, darf aber keine führenden Nullen besitzen. Der Dezimalanteil hat genau zwei Stellen. Der Dezimalanteil kann auch fehlen, dann fehlt auch der Dezimalpunkt.

[16] Dieses Beispiel ist aus Vossen und Witt, [46], pp. 70.

Für diese umgangssprachliche Beschreibung der syntaktisch korrekten Eingaben soll nun eine formale Spezifikation erstellt werden. Für diese Zwecke sind reguläre Ausdrücke die geeigneten Hilfsmittel. Man leitet nun schrittweise und systematisch aus der Prosabeschreibung einen regulären Ausdruck `zahlung` ab, so dass die von `zahlung` generierte Sprache $L(\texttt{zahlung})$ genau die korrekten Ein– und Auszahlungen enthält.

(a) Jedes Wort, *i.e.* jede Zahlung, besteht prinzipiell aus drei Teilen:

- Der Währungsangabe,
- dem Vorzeichen, und
- dem Betrag.

Mit entsprechenden Teilausdrücken — diese werden zu dem Gesamtausdruck konkateniert — `waehrung`, `vorzeichen` und `betrag` gilt dann:

$$\texttt{zahlung} \;=\; \texttt{waehrung} \circ \texttt{vorzeichen} \circ \texttt{betrag}.$$

Sehen wir uns daher die Teilausdrücke der Reihe nach an.

(b) Der reguläre Ausdruck `waehrung` steht für die drei Wörter \$,€ und £, *i.e.* für die Sprache

$$L(\texttt{waehrung}) = \{\$,€,£\}.$$

Diese Sprache wird exakt beschrieben durch den regulären Ausdruck:

$$\texttt{waehrung} = (\$|€|£).$$

(c) Der reguläre Ausdruck `vorzeichen` steht für + bzw. −. Das Vorzeichen kann aber laut Spezifikation auch fehlen. Dies impliziert, dass die Sprache für `vorzeichen` auch das leere Wort enthält. Damit ist

$$L(\texttt{vorzeichen}) = \{+, -, \lambda\}.$$

Diese Sprache wird durch den regulären Ausdruck

$$\texttt{vorzeichen} = (+|-|\lambda)$$

beschrieben.

(d) Der reguläre Ausdruck `betrag` steht für den Betrag, der aus einem ganzzahligen Anteil und dem Dezimalteil besteht. Beide Bestandteile sind durch einen Punkt getrannt. Der Dezimalteil und der Punkt können fehlen. Daher hat der Betrag die Struktur

$$\texttt{betrag} = \texttt{ganz} \circ (\lambda|.\texttt{dez}).$$

Der fehlende Dezimalanteil wird durch das leere Wort λ beschrieben.

(e) Da der ganzzahlige Anteil kein führenden Nullen haben darf, ist er entweder nur 0 oder er beginnt mit einer Ziffer ungleich 0, gefolgt von beliebig vielen Ziffern. Daher:

$$\texttt{ganz} = (0|([1-9] \circ [0-9]^*).$$

Der Dezimalteil besteht aus genau zwei Ziffern, daher

$$\texttt{dez} = [0-9][0-9].$$

(f) Durch Konkatenation der Teilausdrücke erhält man den regulären Ausdruck `zahlung`:

$$\begin{aligned}\texttt{zahlung} = {} & (\$|€|£)\circ \\ & (+|-|\lambda)\circ \\ & (0|([1-9] \circ [0-9]^*)\circ \\ & (\lambda|.[0-9][0-9]).\end{aligned}$$

3.14 Übungen

Übung 3.21:

Geben Sie zu den folgenden regulären Sprachen über dem Alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$ jeweils eine reguläre Grammatik an, die die Sprache erzeugt.

(a)

$$L_1 = \{w \in \Sigma^* \mid w \text{ beginnt mit einer ungeraden Anzahl } \texttt{0}\text{en}, \text{ gefolgt von einer geraden Anzahl } \texttt{1}\text{en}\}.$$

(b) $L_2 = \{w \in \Sigma^* \mid w \text{ enthält mindestens zwei } \texttt{0}\text{en}$ und höchstens eine $\texttt{1}\}$.

(c) $L_3 = \{w \in \Sigma^* \mid w \text{ enthält nicht das Teilwort } \texttt{110}\}$.

Übung 3.22:

Geben Sie für die folgenden Sprachen über dem Alphabet $\Sigma = \{a, b\}$ jeweils einen DFA an, der die jeweilige Sprache akzeptiert.

(a) Die Sprache aller Strings, die genau zwei a enthalten.

(b) Die Sprache aller Strings, die wenigstens zwei a enthalten.

(c) Die Sprache aller Strings, die nicht mit ab enden.

(d) Die Sprache aller Strings, die mit aa oder bb beginnen oder enden.

(e) Die Sprache aller Strings, die nicht den Teilstring aa enthalten.

(f) Die Sprache aller Strings, die eine gerade Anzahl von a und eine gerade Anzahl von b enthalte.

(g) Die Sprache aller Strings, die nicht mehr als einmal den Teilstring aa enthalten. (Der String aaa enthält aa zwei Mal)..

(h) Die Sprache aller Strings, bei denen auf jedes a (falls vorhanden) unmittelbar der Teilstring bb folgt.

(i) Die Sprache aller Strings, die bb und aba als Teilstrings enthalten.

(j) Die Sprache aller Strings, die aba und bab als Teilstrings enthalten.

Übung 3.23:

Geben Sie einen nichtdeterministsichen endlichen Automaten über dem unären Alphabet $\Sigma = \{a\}$ an, der die Sprache

$$L = \{w \in \Sigma^* \mid w = a^n, n \text{ ist durch 3 oder 5 teilbar}\}.$$

erkennt.

Übung 3.24:

Geben Sie zu jeder der folgenden Sprachen einen NFA an (siehe Übung [3.21]), der die Sprache erkennt.

(a)

$$L_1 = \{w \in \Sigma^* \mid w \text{ beginnt mit einer ungeraden Anzahl } \texttt{0}\text{en}, \text{ gefolgt von einer geraden Anzahl } \texttt{1}\text{en}\}.$$

(b) $L_2 = \{w \in \Sigma^* \mid w \text{ enthält mindestens zwei } \texttt{0}\text{en und höchstens eine } \texttt{1}\}$

(c) $L_3 = \{w \in \Sigma^* \mid w \text{ enthält nicht das Teilwort } \texttt{110}\}$.

Leiten Sie aus den Automaten mit dem in dem Beweis von Satz [1] verwendeten Verfahren eine reguläre Grammatik ab. Vergleichen Sie diese mit der Übung [3.21].

Übung 3.25:

Die Sprache

$$L = \{w \in \{a, b\}^* \mid w = (ab)^n a (ba)^n, n \geq 0\}$$

ist regulär.

(a) Geben Sie die Wörter für $n = 0, 1, 2, 3$ an.

(b) Geben Sie einen nichtdeterministischen oder deterministischen endlichen Automaten an, der diese Sprache akzeptiert.

(c) Beschreiben Sie L mit Hilfe eines regulären Ausdrucks.

Übung 3.26:

Betrachten Sie den folgenden `NFA`, der alle Bitstrings akzeptiert, die mit `00` enden.

$$N = (\{q_0, q_1, q_2\}, \{\texttt{0}, \texttt{1}\}, \delta, q_0, \{q_2\})$$

mit der Übergangsrelation δ:

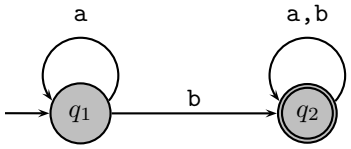
	0	1
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\emptyset$

(a) Konstruieren Sie einen äquivalenten DFA mit Hilfe des Potenzmengenverfahrens.

(b) Konstruieren Sie einen äquivalenten DFA über den Zustandsbaum.

Übung 3.27:

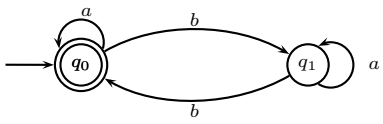
Gegeben ist der folgende DFA.



Konstruieren Sie aus diesem Automaten einen regulären Ausdruck. Verwenden Sie dazu das Verfahren, das im Beweis des Satzes [3.17] angegeben wurde.

Übung 3.28:

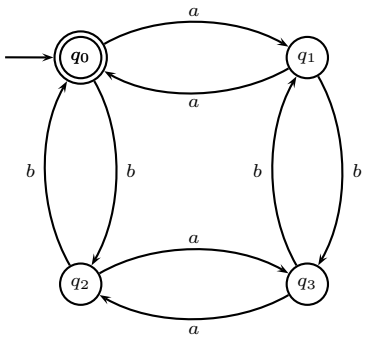
Gegeben ist der folgende DFA, der alle Wörter über dem Alphabet $\Sigma = \{a, b\}$ akzeptiert, die eine gerade Anzahl von bs enthalten.



Konstruieren Sie aus diesem Automaten einen regulären Ausdruck. Verwenden Sie dazu das Verfahren, das im Beweis des Satzes [3.17] angegeben wurde.

Übung 3.29:

Gegeben ist der folgende DFA, der alle Wörter über dem Alphabet $\Sigma = \{a, b\}$ akzeptiert, die eine gerade Anzahl von as und bs enthalten.



Konstruieren Sie aus diesem Automaten einen regulären Ausdruck. Verwenden Sie dazu das Verfahren, das im Beweis des Satzes [3.17] angegeben wurde.

Übung 3.30:

Für einen Binärstring $w \in \{0, 1\}^+$ bezeichne w_2 den Zahlenwert des Binärstrings als Dualzahl interpretiert.

(a) Entwerfen Sie einen deterministischen endlichen Automaten M_1, der alle Binärstrings akzeptiert, die als Dualzahl interpretiert durch 2 teilbar sind. Der Automat soll also die folgende Sprache akzeptieren:

$$L_1 = \{w \in \{0, 1\}^+ \mid w_2 \bmod 2 \equiv 0\}.$$

(b) Entwerfen Sie einen deterministischen endlichen Automaten M_2, der alle Binärstrings akzeptiert, die als Dualzahl interpretiert durch 4 teilbar sind. Der Automat soll also die folgende Sprache akzeptieren:

$$L_2 = \{w \in \{0, 1\}^+ \mid w_2 \bmod 4 \equiv 0\}.$$

Übung 3.31:

Geben Sie jeweils einen nichtdeterministischen endlichen Automaten (NFA) mit der angegeben Anzahl von Zuständen an, der die jeweiligen Sprachen erkennt.

1. Die Sprache $L_1 = \{w \mid w \text{ endet mit } \texttt{00}\}$ mit drei Zuständen.
2. Die Sprache

   $$L_2 = \{w \mid w \text{ enthält den Substring } \texttt{0101}\}$$

   mit fünf Zuständen.
3. Die Sprache

   $L_3 = \{w \mid w \text{ enthält eine gerade Anzahl von } \texttt{0}\text{en oder genau zwei } \texttt{1}\text{en}\}$

   mit sechs Zuständen.
4. Die Sprache $L_4 = \{0\}$ mit zwei Zuständen.
5. Die Sprache $L_5 = \texttt{0}^*\texttt{1}^*\texttt{0}^*\texttt{1}^*$ mit drei Zuständen.
6. Die Sprache $L_6 = \{\lambda\}$ mit einem Zustand.
7. Die Sprache $L_7 = \texttt{0}^*$ mit einem Zustand.

Übung 3.32:

Erstellen Sie die Zustandsdiagramme von deterministischen endlichen Automaten, die die folgenden Sprachen akzeptieren. In allen Fällen ist das zugrundeliegende Alphabet das binäre Alphabet $\{\texttt{0}, \texttt{1}\}$:

1. $\{w \mid w$ beginnt mit einer `1` und endet mit einer `0` $\}$.
2. $\{w \mid w$ enthält mindestens drei `1`en $\}$.
3. $\{w \mid w$ enthält den Substring `0101`, i.e. $w = x\texttt{0101}y$ für x, y $\}$.
4. $\{w \mid w$ hat mindestens die Länge drei und enthält als drittes Symbol die `0` $\}$.
5. $\{w \mid w$ beginnt mit einer `0` und hat ungerade Länge oder beginnt mit einer `1` und hat gerade Länge $\}$.
6. $\{w \mid w$ enthält nicht den Substring `110` $\}$.
7. $\{w \mid$ die Länge von w ist höchstens 5 $\}$.
8. $\{w \mid w$ ist jeder beliebige String mit Ausnahme von `11` und `111` $\}$.
9. $\{w \mid$ jede ungerade Position von w ist eine `1` $\}$.
10. $\{w \mid w$ enthält wenigstens zwei `0`en und höchstens eine `1` $\}$.
11. $\{\lambda, 0\}$.
12. $\{w \mid w$ enthält eine gerade Anzahl von `0`en oder genau zwei `1`en $\}$.
13. Die leere Menge.
14. Alle Strings mit Ausnahme des leeren Strings.

Übung 3.33:

Geben Sie reguläre Ausdrücke an, die die folgenden Sprachen generieren:

1. $\{w \mid w$ beginnt mit einer `1` und endet mit einer `0` $\}$.
2. $\{w \mid w$ enthält mindestens drei `1`en $\}$.
3. $\{w \mid w$ enthält den Substring `0101`, i.e. $w = x\texttt{0101}y$ für x, y $\}$.
4. $\{w \mid w$ hat mindestens die Länge drei und enthält als drittes Symbol die `0` $\}$.
5. $\{w \mid w$ beginnt mit einer `0` und hat ungerade Länge oder beginnt mit einer `1` und hat gerade Länge $\}$.
6. $\{w \mid w$ enthält nicht den Substring `110` $\}$.
7. $\{w \mid$ die Länge von w ist höchstens 5 $\}$.
8. $\{w \mid w$ ist jeder beliebige String mit Ausnahme von `11` und `111` $\}$.
9. $\{w \mid$ jede ungerade Position von w ist eine `1` $\}$.
10. $\{w \mid w$ enthält wenigstens zwei `0`en und höchstens eine `1` $\}$.
11. $\{\epsilon, 0\}$.
12. $\{w \mid w$ enthält eine gerade Anzahl von `0`en oder genau zwei `1`en $\}$.
13. Die leere Menge.
14. Alle Strings mit Ausnahme des leeren Strings.

Übung 3.34:

Ein Spiel ist definiert durch das zweimalige Werfen mit einem Würfel mit den Zahlen 1 bis 6. Das Spiel gilt als gewonnen, wenn die Summe der Augenzahlen durch 3 teilbar ist. Im Folgenden Sei der Ablauf eines Spiels codiert als ein Wort

$$W \in \{1, 2, 3, 4, 5, 6\}^2,$$

wobei das erste Zeichen für den ersten Wurf und das zweite für den zweiten Wurf steht.

(a) Entwickeln Sie eine `DFA` M, welcher die Kodierung eines Spielablaufs genau dann akzeptiert, wenn das Spiel als gewonnen gilt. Geben Sie den Automaten vollständig an.

(b) Minimieren Sie den Automaten M und geben Sie den minimierten `DFA` M' vollständig an.

(c) Verallgemeinern Sie den endlichen Automaten M bzw M', so dass er bei einem beliebig langen Würfelspiel mit einem Würfel genau dann akzeptiert, wenn die Summe der Augenzahlen aller Ürfe durch 3 teilbar ist.

Übung 3.35:

Gegeben ist der reguläre Audruck

$$R = \mathtt{01}(\mathtt{0} + \mathtt{1})^*\mathtt{10}^*.$$

Entwickeln Sie einen nichtdeterministischen endlichen Automaten N und formen Sie diesen anschließend in einen deterministischen endlichen Automaten M um mit $L(R) = L(M) = L(N)$.

1. Verwenden Sie dazu das Potenzmengenverfahren.
2. Verwenden Sie das Verfahren über den Zustandsbaum.

Übung 3.36:

Betrachten Sie die regulären Ausdrücke

$$\begin{aligned}\alpha_1 &= \mathtt{0011}\\ \alpha_2 &= \mathtt{0} + \mathtt{11}\\ \alpha_3 &= \mathtt{01}^*.\end{aligned}$$

Konstruieren Sie zu jedem dieser regulären Ausdrücke einen `NFA` nach dem Konstruktionsverfahren aus dme Kapitel [3.6].

Übung 3.37:

Betrachten Sie die folgende rechtslineare Grammatik G:

$$G = (N, T, P, S)$$

mit $N = \{S, A, B\}, T = \mathtt{0}, \mathtt{1}$ und den Produktionen

$$\begin{aligned}S &\longrightarrow \mathtt{0}S \mid \mathtt{1}S \mid \mathtt{0}A\\ A &\longrightarrow \mathtt{0}B \mid \mathtt{1}B\\ B &\longrightarrow \mathtt{0} \mid \mathtt{1}.\end{aligned}$$

Konstruieren Sie einen äquivalenten `NFA`, indem Sie das Verfahren aus dem Satz [3.9] anwenden.

Übung 3.38:

Betrachten Sie die reguläre Sprache

$L = \{w \in \{\texttt{0},\texttt{1}\}^* \mid w \text{ enthält eine gerade Anzahl von } \texttt{0}\text{en und } \texttt{1}\text{en}\}$.

1. Erstellen Sie einen DFA, der diese Sprache akzeptiert.
2. Konstruieren Sie aus diesem Automaten eine rechtslineare Grammatik, die die Sprache L erzeugt.

Übung 3.39:

Betrachten Sie die reguläre Sprache

$$L = \{w \in \{\texttt{0},\texttt{1}\}^* \mid w \text{ enthält nicht den Teilstring } \texttt{110}\}.$$

1. Konstruieren Sie einen `DFA`, der L akzeptiert.
2. Konstruieren Sie aus dem `DFA` den regulären Ausdruck, der die Sprache L generiert.

Übung 3.40:

Betrachten Sie die folgende Sprache

$$L = \{w \in \{a,b\}^* \mid w \text{ beginnt oder endet mit } aa \text{ oder } bb\}.$$

Zeigen Sie, dass L regulär ist. Konstruieren Sie zu diesem Zweck einen `DFA` mit der in Abschnitt [3.9.1] vorgestellten Methode. Lassen Sie nicht erreichbare Zustände unberücksichtigt.

Übung 3.41:

Minimieren Sie den folgenden `DFA` M:

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{\texttt{0},\texttt{1}\}, \delta, q_0, \{q_4\})$$

mit der Übergangsfunktion

	0	1
q_0	q_1	q_2
q_1	q_4	q_2
q_2	q_4	q_2
q_3	q_4	q_2
q_4	q_0	q_3

Kapitel 4

Kontextfreie Sprachen, Kellerautomaten

Ein wichtiges Kriterium kontextfreier Grammatiken ist, dass auf der linken Seite der Produktionen immer eine einzelne Variable steht. Dadurch werden Ersetzungen während einer Ableitung immer unabhängig vom Kontext vorgenommen. Für die Beschreibung der Syntax von Programmenn reicht diese Form von Grammatiken aus, bis auf einige kontextabhängige Restriktionen. Eine solche ist beispielsweise, dass die Anzahl der formalen Parameter in einem Funktionsaufruf mit der Anzahl der in der Funktionsdefinition vereinbarten Parameter übereinstimmen muss.

Der Teil eines Compilers, der ein eingegebenes Programm auf syntaktische Korrektheit überprüft, ist der **Parser**. Der Parser bekommt vom Scanner eine Folge von Eingabesymbolen und muss feststellen, ob diese korrekt zu Konstrukten wie geschachtelte Klammerausdrücke oder Schleifenanweisungen usw. zusammengesetzt wurden. Prinzipiell geschieht dies dadurch, dass versucht wird, für die Eingabe einen Ableitungsbaum zu konstruieren. Gelingt dies, dann ist das Programm syntaktisch korrekt, andernfalls muss eine geeignete Fehlerbehandlung folgen.

Ohne die Konstruktion des Ableitungsbaums stellt ein Parser einen Erkennungsalgorithmus dar. Das Automatenmodell für die Erkennungsalgorithmen kontextfreier Sprachen sind die **nichtdeterministischen Kellerautomaten**[1] . Im Vergleich mit den endlichen Automaten ist deas Konzept der nichtdeterministischen Kellerautomaten mächtiger als die deterministische Version, so dass nicht zu jeder kontextfreien Sprache ein deterministischer Kellerautomat existiert, der diese Sprache akzeptiert.

Man kann daher nicht für jede kontextfreie Grammatik einen Parser dadurch erhalten, dass man über den Zwischenschritt eine nichtdeterministischen Keller-

[1] Alternativ verwenden wir auch den Begriff Pushdown–Automat

automaten einen deterministischen Kellerautomaten konstruiert, wie es bei der Entwicklung von Scannern auf der Grundlage endlicher Automaten der Fall ist.

Um zu einer beliebigen kontextfreien Grammatik einen Parser zu entwickeln, bleibt einem nichts anderes überig, als alle Alternativen eines nichtdeterministischen Kellerautomaten systematisch durchzuspielen. Diese Art von Parsern ist jedoch sehr ineffizient und für die Praxis ungeeignet. Eine andere Möglichkeit besteht darin, sich auf spezielle kontextfreie Grammatiken zu beschränken, die aber immer noch m
ächtig genug sind, um die wichtigsten syntaktischen Eigenschaften einer Programmiersprache zu beschreiben, und die man in einen effizienten deterministischen Kellerautomaten umwandeln kann.

Wir werden zunächst **Ableitungsbäume** für kontextfreie Grammatiken betrachten und mit ihrer Hilfe die Problematik der *Mehrdeutigkeiten* untersuchen. Diese bilden eine der Hauptursache dafür, dass im allgemeinen Fall zu kontextfreien Grammatiken nur ineffiziente Parser entwickelt werden können. Anschließend untersuchen wir das Modell der Kellerautomaten und sehen uns deren Verwendung als Parsingalgorithmus an. Für kontextfreie Grammatiken gibt es zwei Normalformen:

- die CHOMSKY–Normalform
- und die GREIBACH–Normalform,

die wir uns ebenfalls ansehen. Diese Normalformen spielen bei der Entwicklung von Parsern sowie bei der Herleitung theoretischer Ergebnisse eine wichtige Rolle. Schließlich betrachten wir die Grenzen kontextfreier Grammatiken und zeigen, dass die Sprache

$$L = \{0^n 1^n 2^n \mid n \geq 1\}$$

nicht kontextfrei ist. Dies impliziert, dass die Klasse der kontextfreien Sprachen echt enthalten ist in der Klasse der kontextsensitiven Sprachen.

4.1 Ableitungsbäume

Mit Hilfe von Ableitungsbäumen kann man sich bildhaft klarmachen, wie ein Wort — ausgehend vom Startsymbol — abgleitet wird. Da bei kontextfreien Grammatiken bei jedem Ableitungsschritt immer genau eine Variable durch eine Symbolfolge ersetzt wird, kann man an diese Variable für jedes neue Symbol eine Verzweigung nach unten zeichnen. Auf diese Weise erhält man eine Baumstruktur mit dem Startsymbol als Wurzel und den Terminalen des abgleiteten Wortes als Blätter.

Beispiel [4.59]

Gegeben ist die folgende kontextfreie Grammatik

$$G = (N, T, P, S)$$

mit $N = \{S\}$,

$$T = \{\times, +, (,), a\}$$

und den Produktionen P:

$$\begin{array}{lcl} S & \longrightarrow & S + S \\ S & \longrightarrow & S \times S \\ S & \longrightarrow & (S) \\ S & \longrightarrow & a \end{array}$$

Die von G erzeugt Sprache besteht aus allen verschachtelten Klammerausdrücken, die mit den beiden Operatoren $\times$ und $+$ und dem Zeichen a gebildet werde. a ist ein Platzhalter für Zahlen oder Bezeichner.

Betrachten wir das Wort $w = a + (a + a) \times a$. Eine Ableitung dieses Wortes sieht wie folgt aus:

$$\begin{array}{rcl} S & \Longrightarrow & S + S \\ & \Longrightarrow & S + S \times S \\ & \Longrightarrow & S + (S) \times S \\ & \Longrightarrow & S + (S + S) \times S \\ & \Longrightarrow & a + (S + S) \times S \\ & \Longrightarrow & a + (a + S) \times S \\ & \Longrightarrow & a + (a + a) \times S \\ & \Longrightarrow & a + (a + a) \times a \end{array}$$

Diese Ableitung lässt sich durch den Ableitungsbaum in der Abbildung [4.1] darstellen.

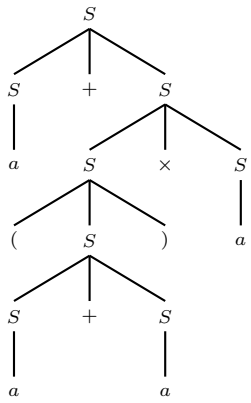


Abbildung 4.1: Ableitungsbaum

Definition [4.20]:

Sei

$$G = (N, T, P, S)$$

eine kontextfreie Grammatik. Ein **Ableitungsbaum** oder **Parserbaum, Strukturbaum, Syntaxbaum** oder **Parsetree** für G ist ein gerichteter, geordneter Baum mit Markierungen, wobei gilt:

1. Die Blätter sind mit Terminalzeichen $a \in T$ markiert.
2. Die inneren Knoten sind mit Variablen $X \in N$ markiert.
3. Ist A die Markierung eines inneren Knotens und sind $X_1, X_2, \ldots, X_n$ die Markierungen der direkten Nachfolger, so gibt es eine Produktion
$$A \longrightarrow X_1 X_2 \ldots X_n.$$

4.2 Mehrdeutigkeiten

Bei der Überprüfung der syntaktischen Korrektheit eines Programmkonstrukts versucht der Parser nachzuweisen, dass die Eingabe in der zugehörigen kontextfreien Grammatik abgeleitet werden kann, Im Prinzip erfolgt dies durch die Konstruktion eines Parserbaums für die Eingabe.

Bei einer kontextfreien Grammatik kann es für ein Wort der erzeugten Sprache im Allgemeinen mehrere verschiedene Ableitungsbäume geben, dies zeigt das folgende Beispiel für das Wort $a + a \times a$ in der Grammatik aus dem Beispiel [4.59]:

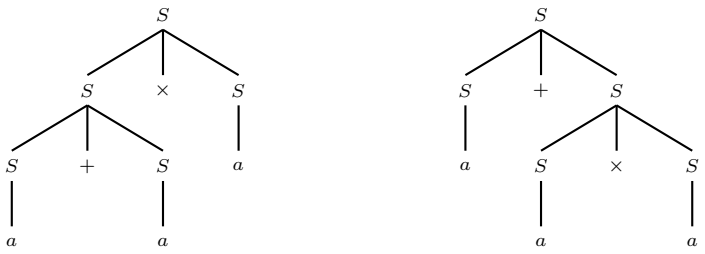


Abbildung 4.2: Mehrdeutigkeiten beim Anleiten eines Wortes.

Natürlich sollte die Konstruktion eines Ableitungsbaums durch den Parser in eindeutiger Weise erfolgen können, was bei der Grammatik dieses Beispiels nicht möglich ist. Ein Parser, der auf dieser Grammatik basiert, kann offensichtlich nicht entscheiden, ob zuerst multiplisziert und dann addiert wird oder umgekehrt, *i.e.* die *Punkt–vor–Strich* Regel kann nicht realisiert werden.

Wir betrachten das Problem der Mehrdeutigkeiten in Ableitungen etwas genauer, insbesondere interessieren die Zusammenhänge zwischen Ableitungen und Ableitungsbäumen.

Beispiel [4.60]

Wir betrachten die kontextsensitive Grammatik

$$G = (N, T, P, S)$$

mit $N = \{S\}, T = \{a, b\}$, Startvariable S und den drei Produktionen P:

$$S \longrightarrow SS,$$
$$S \longrightarrow a,$$
$$S \longrightarrow b.$$

Das Wort aba kann in G folgendermaßen abgleitet werden:

$$\left.\begin{array}{rcl} S & \Longrightarrow & SS \\ & \Longrightarrow & SSS \\ & \Longrightarrow & aSS \\ & \Longrightarrow & abS \\ & \Longrightarrow & aba \end{array}\right\} \tag{4.1}$$

Diese Ableitung können wir durch die beiden folgenden Ableitungsbäume repräsentieren:

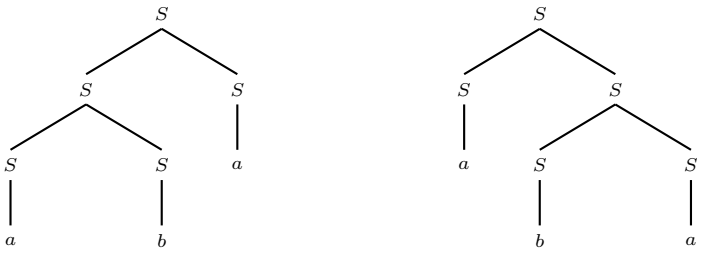


Abbildung 4.3: Zur Mehrdeutigkeit in Ableitungen.

Offensichtlich kann aus der Ableitung nicht in eindeutiger Weise abgelesen werden, welches der beiden S in dem Schritt $SS \Longrightarrow SSS$ durch SS ersetzt wurde. Um dieses Problem zu umgehen, legt man sich auf eine Reihenfolge bei der Ersetzung der Variablen fest. Entweder wird immer die ganz links stehende Variable, oder die am weitesten rechts stehende Variable ersetzt.

Definition [4.21]:

Eine Ableitung in einer kontextfreien Grammatik G heißt **Linksableitung**, wenn immer die am weitesten links stehende Variable ersetzt wird. Wird immer die am weitesten rechts stehende Variable ersetzt, so spricht man von einer **Rechtsableitung**.

Es gilt nun, dass jede Linksableitung durch genau einen Ableitungsbaum repräsentiert wird, und jedem Ableitungsbaum entspricht in eindeutiger Weise

eine Linksableitung. In gleicher Weise entsprechen sich Rechtsableitungen und Ableitungsbäume eineindeutig.

Die Ableitung (4.1) in dem Beispiel [4.60] ist eine Linksableitung. Sie wird in eindeutiger Weise durch den linken Ableitungsbaum der Abbildung [4.3] repräsentiert.

Das Problem der mehrdeutigkeit wird durch die Festlegung der Abarbeitungsreihenfolge etwas eingegrenzt, jedoch nicht behoben.

Beispiel [4.61]

Betrachte die Grammatik

$$G = (\{S\}, \{a, b\}, P, S))$$

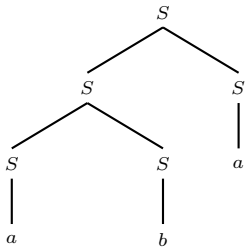
mit den Produktionen

$$\begin{array}{ll} S & \longrightarrow SS, \\ S & \longrightarrow a, \\ S & \longrightarrow b. \end{array}$$

Wir leiten das Wort aba in dieser Grammatik ab:

$$\begin{array}{rl} S & \Longrightarrow SS \\ & \Longrightarrow SSS \\ & \Longrightarrow aSS \\ & \Longrightarrow abS \\ & \Longrightarrow aba. \end{array}$$

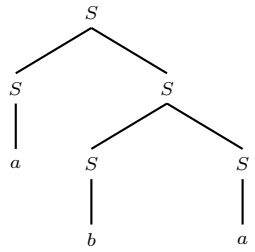
Der Ableitungsbaum ist:



Andererseits können wir auch die folgende Ableitung durchführen:

$$\begin{array}{rcl} S & \Longrightarrow & SS \\ & \Longrightarrow & aS \\ & \Longrightarrow & aSS \\ & \Longrightarrow & abS \\ & \Longrightarrow & aba. \end{array}$$

Der Ableitungsbaum für diese Ableitung ist:



Definition [4.22]:

Eine kontextfreie Grammatik G heißt **mehrdeutig**, falls es ein Wort w in $L(G)$ gibt, das mindestens zwei verschiedene Ableitungsbäume (bzw. Linksableitungen oder Rechtsableitungen) hat. Andernfalls heißt G eindeutig.

Eine kontextfreie Sprache wenn alle Grammatiken, die L erzeugen, mehrdeutig sind. Andernfalls nennt man L eindeutig.

Damit eine Sprache als eindeutig charakterisiert werden kann, genügt es also, eine eindeutige Grammatik für die Sprache zu finden. Dass eine Sprache inhärent mehrdeutig ist, ist schwieriger zu zeigen, denn man muss in diesem Fall *alle* Grammatiken für diese Sprache betrachten.

Beispiel [4.62]

Die Grammatik aus dem Beispiel [4.61] ist mehrdeutig, denn für das Wort *aba* haben wir zwei verschiedene Linksableitungen angegeben.

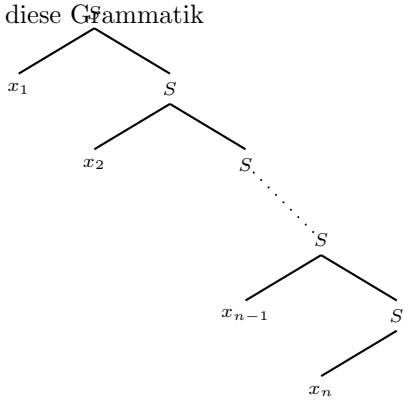
Betrachte die Grammatik

$$G = (\{S\}, \{a, b\}, P, S)$$

und den Produktionen

$$\begin{array}{lcl} S & \longrightarrow & aS, \\ S & \longrightarrow & bS, \\ S & \longrightarrow & a, \\ S & \longrightarrow & b. \end{array}$$

Diese Grammatik erzeugt die gleiche Sprache und ist eindeutig, denn alle Ableitungsbäume für diese Grammatik



Für jedes Wort $w = x_1 x_2 \dots x_{n-1} x_n \in L(G)$ kann es immer nur einen Ableitungsbaum geben Folglich ist die Sprache

$$L = L(G) = \{a, b\}^+$$

eindeutig.

Beispiel [4.63]

Alle regulären Sprachen sind eindeutig. Denn die Konstruktion einer regulären Grammatik aus einem deterministischen endlichen Automaten liefert immer eine eindeutige Grammatik, da sie das deterministische Abarbeiten des Inputs durch entsprechende Linksableitungen modelliert.

Beispiel [4.64]

Eine typische inhärent mehrdeutige Sprache ist

$$L = \{a^i b^j c^k \mid i, j, k \geq 1, i = j \text{ oder } j = k\}.$$

Eine Grammatik, die diese Sprache generiert ist:

$$G = (N, T, P, S)$$

mit

$$\begin{aligned} N &= \{S, A, C, X, Y\} \\ T &= \{a, b, c\}, \\ S &= \text{ Startvariable}, \end{aligned}$$

und den Produktionen P:

$$\begin{array}{rcl} S & \longrightarrow & XC \mid AY \\ X & \longrightarrow & aXb \mid ab \\ C & \longrightarrow & cC \mid c \\ Y & \longrightarrow & bYc \mid bc \\ A & \longrightarrow & aA \mid a. \end{array}$$

man kann zeigen, dass jede kontextfreie Grammatik, die diese Sprache erzeugt, mehrdeutig ist. In der obigen Grammatik hat das Wort

$$w = aabbcc$$

die folgenden beiden Ableitungsbäume.

Mehrdeutigkeiten in Programmiersprachen

Die Frage, ob eine gegebene kontextfreie Grammatik mehrdeutig ist, ist unentscheidbar.[2] Daher muss jede Grammatik individuell auf Mehrdeutigkeiten untersucht werden, um diese zu vermeiden. Im Folgenden sehen wir uns zwei typische Mehrdeutigkeitsprobleme an, die bei Programmiersprachen auftreten.

I: Arithmetische Ausdrücke

Wir haben im letzten Abschnitt gesehen, dass die kontextfreie Grammatik

$$G = (\{S\}, \{a, +, \times, (,)\}, P, S)$$

mit den Produktionen P:

$$\begin{array}{rcl} S & \longrightarrow & S + S, \\ S & \longrightarrow & S \times S, \\ S & \longrightarrow & (S), \\ S & \longrightarrow & a \end{array}$$

eine eingeschränkte Form von arithmetischen Ausdrücken beschreibt. Diese Grammatik ist mehrdeutig, da es zum Beispiel zwei verschiedene Ableitungsbäume für den Ausdruck $a + a \times a$ in G gibt. Mit dieser Grammatik kann die *Punkt–vor–Strich* Regel nicht umgesetzt werden.

Wir geben nun eine kontextfreie Grammatik G' an für die gleiche Sprache, die eindeutig ist. Die Ableitungsbäume dieser Grammatik realisieren stets die *Punkt–vor–Strich* Regel. Wir haben

$$G' = (N', T', P', S')$$

[2] Einen Beweis findet man in Wegener.

mit den Variablen

$$N' = \{Ausdruck, Term, Faktor\},$$

dem Terminalalphabet

$$T' = \{+, \times, (,), a\},$$

der Startvariablen

$$S' = Ausdruck$$

und den Produktionen P' mit

$$\left.\begin{array}{rcl} Ausdruck & \longrightarrow & Ausdruck + Term \\ Ausdruck & \longrightarrow & Term \\ Term & \longrightarrow & Term \times Faktor \\ Term & \longrightarrow & Faktor \\ Faktor & \longrightarrow & (Ausdruck) \\ Faktor & \longrightarrow & a. \end{array}\right\} \tag{4.2}$$

Diese Grammatik beschreibt die Tatsache, dass ein arithmetischer Ausdruck grundsätzlich zwei Ebenen für Verknüpfungen aufweist. Die obere Ebene für die Strich–Operationen (Addition und Subtraktion), die untere Ebene für die Punkt–Operationen (Multiplikation, Division). Diese beiden Ebenen können sich in beliebiger Tiefe verschachteln, was durch geeignete Klammerung organisiert ist.

Die Subtraktion und die Division können in die Grammatik integriert werden, indem man die beiden Produktionen

$$\begin{array}{rcl} Ausdruck & \longrightarrow & Ausdruck - Term, \\ Term & \longrightarrow & Term/Faktor \end{array}$$

hinzufügt.

In G' hat das Wort $a + a \times a$ den in der Abbildung [4.4] dargestellten eindeutigen Ableitungsbaum.

II: if–else Mehrdeutigkeiten

In allen Programmiersprachen gibt es if– bzw. if–else Anweisungen, für die üblicherweise die folgenden Regeln gelten:

$$\begin{array}{rcl} stmt & \longrightarrow & \texttt{if}\ (expr)\ stmt \\ & & \mid\ \texttt{if}\ (expr)\ stmt\ \texttt{else}\ stmt \\ & & \mid\ other\_stmt \end{array}$$

Hier steht *stmt* für *statement*, *expr* für *expression*, die Variable *other_stmt* steht für Zuweisung, Block oder Schleife.

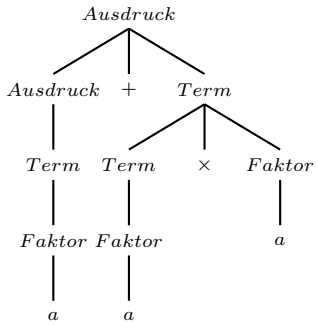
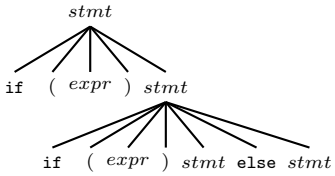


Abbildung 4.4: Ableitungsbaum für das Wort $a + a \times a$.

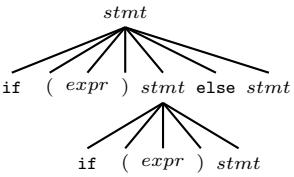
Das obige Grammatikfragment ist mehrdeutig, denn für den Ausdruck

`if` (*expr*) `if` (*expr*) *stmt* `else` *stmt*

lassen sich in der obigen Grammatik zwei verschiedene Ableitungsbäume angeben.



und



Diese Mehrdeutigkeit ist unter dem Begriff **dangling else** bekannt. Üblicherweise wird der `else`–Zweig an das am nächsten davor stehende noch freie `if` gebunden. Das ist die Lesart, die durch den oberen Ableitungsbaum repräsentiert wird.

Die Eigenschaft, dass ein `else` immer an das am nächsten davor stehende noch freie `if` gebunden wird, kann dadurch erreicht werden, dass zwischen

einem `if` und einem `else` immer nur eine sogenannte *geschlossene Anweisung*[3] erlaubt ist. In einer solchen Anweisung ist ein ungebundenes `if` nicht zulässig. Die folgende Grammatik fasst diese Bedingung in Regeln und wird dadurch zu einer eindeutigen Grammatik.

$$
\begin{array}{lcl}
stmt & \longrightarrow & matched\_stmt \mid unmatched\_stmt \\
matched_s tmt & \longrightarrow & \texttt{if}\,(expr)\,matched\_stmt\,\texttt{else}\,matched\_stmt \\
 & & \mid other\_stmt \\
unmatched_s tmt & \longrightarrow & \texttt{if}\,(expr)\,stmt \\
 & & \mid \texttt{if}\,(expr)\,matched\_stmt\,\texttt{else}\,unmatched\_stmt
\end{array}
$$

Mit diesen Regeln kann die Anweisung

$$\texttt{if}\ (expr)\ \texttt{if}\ (expr)\ other\_stmt\ \texttt{else}\ other_s tmt$$

in eindeutiger Weise mit der durch den Ableitungsbaum in Abbildung [4.5] dargestellten Linksableitung abgeleitet werden.

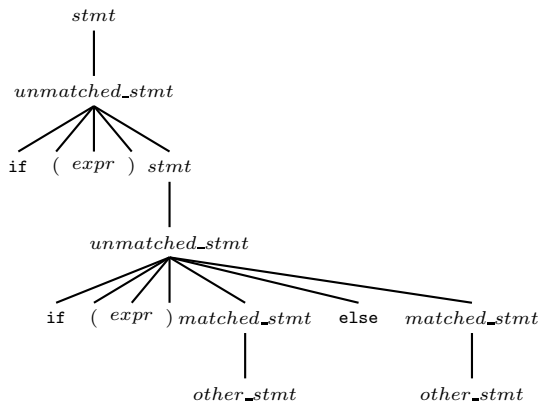


Abbildung 4.5: Ableitungsbaum

Es ist nachvollziehbar, dass in Lehrbüchern über Programmiersprachen üblicherweise die beiden Regeln der mehrdeutigen Grammatik angegeben sind. Diese zeigne sehr einfach, welche beiden `if` Anweisungen mit welcher Syntax zur Verfügung stehen. Die eindeutige Grammatik ist nicht sehr anschaulich und wird nur bei der Entwicklung von Parsern eingesetzt.

[3] Engl.: *matched statement.*

4.3 Top–Down Analyse

Für den Aufbau eines Ableitungsbaums durch den Parser gibt es unterschiedliche Strategien. Wir führen als Beispiel eine Top–Down Analyse durch, bei der der Baum mit der Wurzel beginnend von oben nach unten[4] konstruiert wird. Die Eingabe wird von links nach rechts gelesen. Die verwendete Parsingstrategie nennt man *Syntaxanaylse durch rekursiven Abstiegt*. Es gibt verschiedene Parsingstrategien, darunter auch wesentlich effektivere als die Syntaxanalyse durch rekursiven Abstieg. Für unsere Zwecke ist diese hier besser geeignet, da diese Strategie direkt zu dem Kellerautomaten führt.

Wir wollen den arithmetischen Ausdruck

$$a + a \times a$$

analysieren. Dieser Ausdruck ist ein Wort in der Sprache, die die eindeutige Grammatik erzeugt, deren Produktionen durch Gl. (4.2) gegeben sind. Die Produktionen (4.2) enthalten die beiden Regeln:

$$\begin{aligned} Ausdruck &\longrightarrow Ausdruck + Term, \\ Term &\longrightarrow Term \times Faktor \end{aligned}$$

Diese beiden Regeln sind **linksrekursiv** und wegen dieser Eigenschaft nicht für die Parsingstrategie geeignet.[5]

Definition [4.23]:

> Eine kontextfreie Grammatikregel heisst **linksrekursiv**, wenn die Variable der linken Seite als erstes Zeichen auf der rechten Seite vorkommt. Kommt die Variable der linken Seite als letztes Zeichen der rechten Seite vor, so nennt man die Regel **rechtsrekursiv**.

Das Problem läßt sich dadurch lösen, dass man die linksrekursiven Regeln durch rechtsrekursive ersetzt. Dies ist immer möglich, *i.e.* es gibt einen Algorithmus, der bescheibt, wie man jede linksrekursice Regel ersetzen kann ohne die generierte Sprache zu ändern.[6]

Für eine übersichtlichere Darstellung setzen wir:

$$\begin{aligned} Ausdruck &\longleftrightarrow A, \\ Term &\longleftrightarrow T, \\ Faktor &\longleftrightarrow F. \end{aligned}$$

[4] Man beachte, dass in der Informatik die Bäume stets von oben nach unten wachsen.
[5] Die linksrekursiven Regeln führen zu Endlosschleifen, siehe dazu [52]. Kapitel 2.4.
[6] Dies findet man in [52].

Die folgende kontextfreie Grammatik ist eine eindeutige Grammatik ohne Linksrekursion für eingeschränkte arithmetische Ausdrücke:

$$\left.\begin{array}{lcl} A & \longrightarrow & T + A \mid T \\ T & \longrightarrow & F \times T \mid F \\ F & \longrightarrow & (A) \mid a \end{array}\right\} \tag{4.3}$$

Der Parser soll das Wort $a + a \times a$ als Input erhalten und dazu den Ableitungsbaum der Abbildung [4.6] konstruieren.

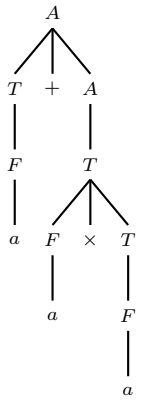


Abbildung 4.6: Ableitungsbaum des arithmetischen Ausdrucks $a + a \times a$ in der Grammatik (4.3).

Wir setzen voraus, dass der Parsealgorithmus bei der Auswahl der Grammatikregeln, mit denen der Ableitungsbaum erstellt wird, immer diejenige Regel verwendet, die zum Erfolg führt. Dass dieser Punkt nicht trivial ist, diskutieren wir später.

Beim Parsing wird das Eingabewort Zeichen für Zeichen von links nach rechts gelesen. Jedes Zeichen wird mit dem am weitesten links stehenden noch nicht bearbeiteten Blatt des bisherigen Ableitungsbaums verglichen wird. Ist das Blatt eine Variable, so wird der Baum an dieser Stelle nach links in der Tiefe solange erweitert, bis in den neu entstandenen Blättern an erster Stelle ein terminales Symbol steht. Ist dieses Terminalzeichen verschieden vom Eingabezeichen, erfolgt ein Abbruch. Stimmt das Terminalzeichen mit dem Inputzeichen überein, dann wird das nächste Zeichen der Eingabe mit dem nächsten Blatt des Ableitungsbaums verglichen und der Vorgang solange wiederholt, bis die Übereinstimmung des Eingabewortes mit dem Wort in den Blättern feststeht.

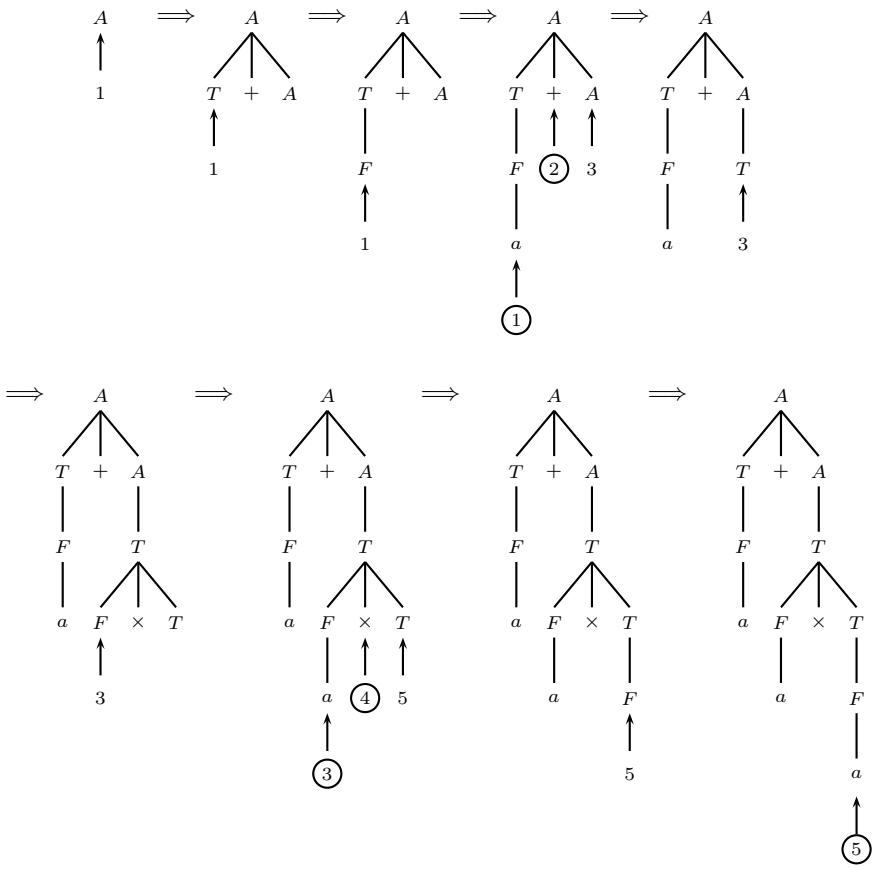


Abbildung 4.7: Aufbau eines Parsetrees beim Lesen des Inputstrings $a + a \times a$.

Die Abbildung [4.7] zeigt die einzelnen Schritte des Parsing Prozesses. Die gelesene Stelle des Eingabewortes wird durch einen Zeiger markiert, der beim Weiterrücken jeweils die Nummer des gerade gelesenen Felds erhält.

Inputstring	a	$+$	a	$\times$	a	
	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
	1	2	3	4	5	6

Im Ableitungsbaum zeigt ein Zeiger auf die Stelle, die gerade bearbeitet wird. Die Nummer entspricht dem Eingabefeld, welches gerade gelesen wird. Dieser Zeiger befolgt beim Weiterrücken immer die Regel

$$Tiefe\ zuerst,\ von\ links\ nach\ rechts.$$

Beim Aufbeu des Ableitungsbaums sind drei Szenarien zu unterscheiden, von denen jeweils der folgende Schritt abhängt.

- Zeigt der Baumzeiger auf eine Variable, dann wird der Baum an dieser Stelle unter Verwendun einer Produktion expandiert. Der Baumzeiger setzt sich auf den ersten direkten Nachfolger.
- Zeigen sowohl der Eingabezeiger als auch der Baumzeiger auf das gleiche Terminalsymbol, dann rücken beide Zeiger eine Stelle weiter. Dieser Fall ist in der Skizze [4.7] durch eine umkreiste Nummer bei dem Baumzeiger dargestellt.
- Zeigen der Baumzeiger und der Eingabezeiger auf verschiedene Terminalzeichen, erfolgt ein Abbruch.

Hat der Baumzeiger den Ableitungsbaum vollständig durchlaufen und ist der Eingabezeiger hinter dem Input angelangt, dann bedeutet dies, dass die Blätter des Ableitungsbaums in der Reihenfolge von links nach rechts genau das Eingabewort enthalten. In diesem Fall wurde also ein passender Ableitungsbaum gefunden.

Das obige Beispiel zeigt einen erfolgreichen Verlauf, bei dem bei der Expansion eines Knotens immer die zum Erfolg führende Regel verwendet wird. Es kann vorkommen, dass zur Expansion eines Knotens mehrere Grammatikregeln angewendet werden können. Im obigen Beispiel hat die Startvariable zwei Regeln,

$$A \longrightarrow T + A \mid T,$$

i.e. wir hätten die Wurzel des Baums zu einem Term T expandieren können. Es ist sehr schnell zu erkennen, dass dann sämtliche Versuche, einen Parsetree für die Eingabe zu erhalten erfolglos sind.

Da ein Parser nicht erkennen kann, welche Regel zum Erfolg führt, wird eine Strategie ausgewählt, welche Regel verwendet wird. Gibt es mehrere Alternativen für die Anwendung einer Regel, merkt sich der Parser diese Alternativen und fährt solange fort, bis er Erfolg hat, oder in eine Sackgasse gerät. Beim Misserfolg eines Versuchs setzt er das Verfahren zur letzten Stelle zurück, an der es eine noch nicht berücksichtigte Möglichkeit zur Anwendung einer Regel gibt, und startet einen neuen Versuch. Dazu wird der ab dieser Stelle erzeugt Teilbaum gelöscht und der Zeiger auf der Eingabe entsprechend zurückgesetzt werden. Das systematische Rücksetzen zu noch nicht berücksichtigten Alternativen nennt man **Backtracking**.

In [52], Seite 216ff. wird beschrieben, wie durch **Linksfaktorisierung** das Problem des Zurücksetzens behoben werden kann. Dabei werden die für eine Expansion in Frage kommenden Regeln so umgeformt, dass die Entscheidung, welche Regel verwendet werden soll, hinausgeschoben werden kann, bis der weitere Verlauf eine eindeutige Auswahl ermöglicht. Diese Methode wird in sogenannten **prädikativen Parsern** eingesetzt, bei denen für die Entscheidung, welche Regel angewendet wird, ein Eingabesymbol nach vorne geschaut wird.

Damit wird auch verständlich, warum linksrekursive Grammatikregeln für die oben beschrieben Parsingstrategie ungeeignet sind. Sie führen den Parser in

eine Endlosschleife. Hat der Parser eine Variable mit einer linksrekursiven Regel expandiert, dann steht diese Variable wieder an erster Stelle der Nachfolger und muss im nächsten Schritt expandiert werden. Da der Parser inzwischen keine weitere Informationen hinzugewonnen hat, wählt er aus dem gleichen Grund wie im Schritt zuvor wieder diese Regel. Die wiederholt sich endlos.

Das gleiche Problem tritt auf, wenn sich die Linksrekursion nicht direkt, sondern über mehrere Stufen hinweg ergibt. Auch diese **indirekte Linksrekursion** kann durch ein systematisches Verfahren[7] beseitigt werden. Ein spezieller Typ indirekter Linksrekursion sind Zyklen von Variablenumbenennungen, diese nennt man **Kettenregeln**, beispielsweise

$$\begin{array}{lcl} X & \longrightarrow & Y \\ Y & \longrightarrow & Z \\ Z & \longrightarrow & X. \end{array}$$

[7] Siehe [52].

4.4 Parser als Erkennungsalgorithmen

Parsingalgorithmen als Teil des Compilers haben die Aufgabe, die syntaktische Korrektheit der Eingabe zu entscheiden und bei Erfolg einen Ableitungsbaum für die Übersetzung zu erstellen. Ignoriert man die Konstruktion des Baums, dann stellt ein Parsingalgorithmus einen Erkennungsalgorithmus für eine Sprache dar.

Die Steuerung eines solchen Erkennungsalgorithmus kann mit Hilfe eines **Kellerspeichers** erfolgen. Synonym verwendet man auch die Begriffe **Stapelspeicher** oder kurz **Stack**. Der Zugriff auf diesen Speicher erfolgt ausschließlich durch das LIFO Prinzip.[8] Das bedeutet, auf den Speicher kann nur auf einer Seite etwas hineingeschrieben werden, dies wird durch die `push`–Operation ausgeführt. Gelesen kann nur das oberste Speicherelement, diese Operation nennt man `pop`–Operation.

Anschaulich kann man sich einen Kellerspeicher als senkrechter Stapel vorstellen,[9] der nur von oben bedient werden kann.

Die Abbildung [4.8] zeigt das Schema eines endlichen Automaten. Die Kontrolleinheit repräsentiert die Zustände und die Übergangsfunktionen, das Band entspricht dem Inputstring und der Pfeil stellt einen Lesekopf dar, der auf das nächste zu lesende Inputsymbol zeigt.

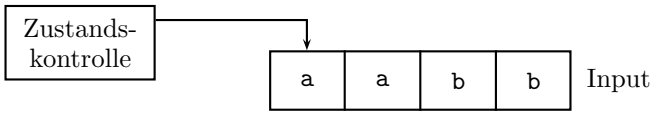


Abbildung 4.8: Schema eines endlichen Automaten.

Fügt man diesem endlichen Automaten nun eine Stackkomponente hinzu, erhält man das Schema eines Kellerautomaten wie in Abbildung [4.9] dargestellt.

Ein Kellerautomat kann Symbole auf den Stack schreiben und diese irgendwann später wieder lesen. Beim Schreiben eines Symbols auf den Stack verschiebt sich der Stackinhalt komplett nach unten (kurz: *pushdown*). Das Symbol, das an oberster Stelle des Stacks abgelegt ist, kann jederzeit wieder gelesen werden

[8] LIFO = Last in First Out

[9] Der Tellerspender in der Essensausgabe der Mensa ist eine sehr gute Metapher für einen Stackspeicher.

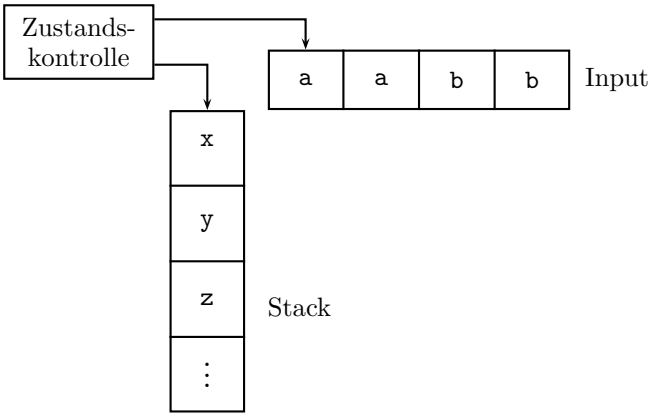


Abbildung 4.9: Schema eines Kellerautomaten.

und wird durch das Lesen vom Stack entfernt. Die auf dem Stack verbleibenden Symbole rutschen dadurch eine Position nach oben. Man nennt häufig das Ablegen eines Symbols auf den Stack eine **Push–Operation**, das Entfernen eines Symbols nennt man **Pop–Operation**. Man beachte, dass sämtliche Schreib– und Lesezugriffe auf den Stack ausschließlich auf die oberste Speicherstelle geschehen. Mit anderen Worten, ein Stack ist eine *last in first out* Speichereinheit. Werden bestimmte Informationen auf dem Stack abgelegt und werden später weitere Informationen in den Stack geschrieben, kann auf die zuerst gespeicherten Informationen erst dann wieder zugegriffen werden, wenn die zuletzt abgelegten Informationen vom Stack weggenommen werden.

Ein bekanntes Beispiel für einen Stack ist ein Tellerstapel, wie man ihn in der Mensa sieht. Unter den Tellern befindet sich eine Feder mit ausreichender Spannung, um gerade einen Teller über der Kante der Theke erscheinen zu lassen. Wird dieser oberste Teller entfernt, wird das Gewicht auf der Feder leichter und der Teller darunter erscheint über der Thekenkante. Wird ein Teller auf den Stapel gestellt, dann wird der Stapel nach unten gedrückt und nur der neu aufgelegte Teller ist sichtbar. Für unsere Zwecke machen wir die idealisierte Annahme, dass die Feder beliebig lang ist, so dass beliebig viele Teller auf dem Stapel abgelegt werden können. Mit dieser Annahme kann also ein Stackspeicher eine unbegrenzte Anzahl an Informationen speichern.

Ansteatt einen vollständigen Ableitungsbaum zu generieren, werden bei einem Erkennungsalgorithmus im Stack lediglich diejenigen Knoten des bisher erzeugten Ableitungsbaums gespeichert, die noch bearbeitet werden müssen. Wird ein Knoten nicht mehr benötigt, wird er gelöscht.

Der Stackinhalt wird (für unsere momentanen Zwecke) wie folgt erzeugt.

Regel 1: Zu Beginn steht nur das Startsymbol (der Grammatik) auf dem Stack.

Regel 2: Liegt eine Variable X oben auf dem Stack, wird eine passende Produktion $X \longrightarrow \alpha$ gesucht und X auf dem Stack durch den String α ersetzt ($\alpha \in (N \cup T)^+$).

Regel 3: Liegt ein Terminalzeichen oben auf dem Stack und stimmt dieses mit dem aktuellen Zeichen des Inputstrings überein, dann wird das oberste Stackzeichen gelöscht (*i.e.* einfach vom Stack weggenommen mittels `pop`–Operation) und der Zeiger der Eingabe in Feld weitergerückt.

Regel 4: Liegt ein Terminalzeichen oben auf dem Stack, das von dem aktuellen Inputzeichen verschieden ist, dann erfolgt ein Abbruch. Im Fall des Abbruchs müssen durch Backtracking alle noch offenen Möglichkeiten durchgespielt werden.

Regel 5: Nach der erfolgreichen Abarbeitung eines Inputstrings ist der Stackspeicher leer.

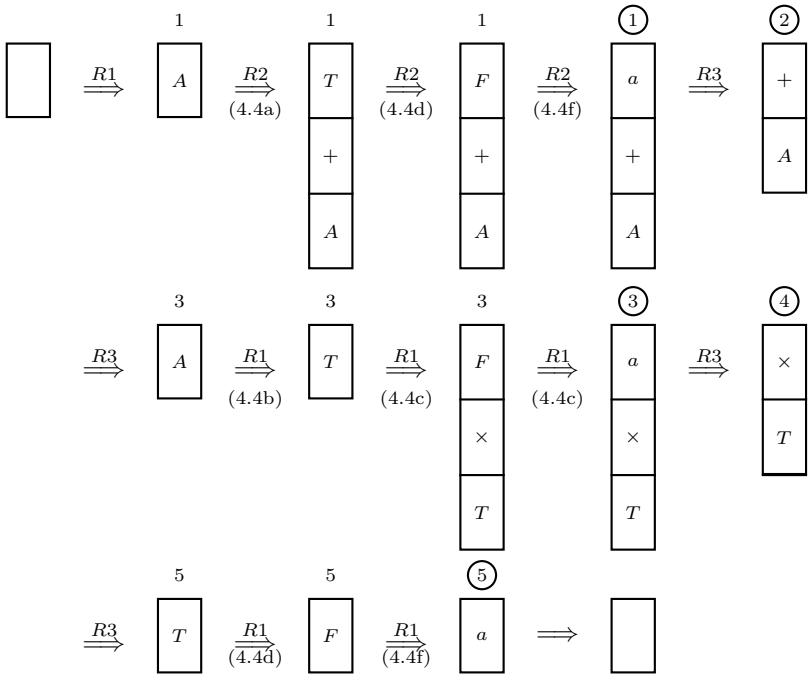
Beispiel [4.65]

Wir demonstrieren diesen Algorithmus am Beispiel des letzten Abschnitts, *i.e.* wir betrachten die Verarbeitung des Wortes $a + a \times a$ durch einen Parser als Erkenner. Wir betrachten die Grammatik

$$\begin{aligned}
A &\longrightarrow T + A && \text{(4.4a)}\\
A &\longrightarrow T && \text{(4.4b)}\\
T &\longrightarrow F \times T && \text{(4.4c)}\\
T &\longrightarrow F && \text{(4.4d)}\\
F &\longrightarrow (A) && \text{(4.4e)}\\
F &\longrightarrow a && \text{(4.4f)}
\end{aligned}$$

Inputstring	a	$+$	a	$\times$	a	
	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
	1	2	3	4	5	6

Über dem Stackinhalt steht jeweils die Nummer der Stelle, auf der sich der Eingabezeiger befindet.



Aus der Abbildung — wie der Stack bearbeitet wird — erkennt man, dass der Parser durch das Lesen des Inputs von links nach rechts auf dem Stack die folgende Linksableitung nachvollzieht:

$$\begin{aligned} A &\Longrightarrow T + A \\ &\Longrightarrow F + A \\ &\Longrightarrow a + A \\ &\Longrightarrow a + T \\ &\Longrightarrow a + F \times T \\ &\Longrightarrow a + a \times T \\ &\Longrightarrow a + a \times F \\ &\Longrightarrow a + a \times a \end{aligned}$$

Das Beispiel [4.65] macht deutlich, dass es eine enge Beziehung gibt zwischen dem Erkennungsalgorithmus einerseits und der zugrundliegenden kontextfreien Grammatik andererseits. In der Tat ist das Beispiel [4.65] ein erstes Beispiel eines **Kellerautomaten** oder auch **Pushdown–Automaten**. Kellerautoma-

ten dienen als Modell für das Erkennen kontextfreier Sprachen. Sie bilden die Grundlagen von Parsingalgorithmen für Programmiersprachen.

4.5 Arbeitsweise von Pushdown–Automaten

Die prototypische Sprache, die sich mit Kellerautomaten beschreiben lässt, ist die formale Sprache

$$L = \{w \in \{0,1\}^* \mid \mathtt{w} = \mathtt{0^n1^n}, \mathtt{n} \geq \mathtt{1}\}. \tag{4.5}$$

Diese Sprache abstrahiert die Eigenschaft, dass zwei Symbole in gleicher Anzahl vorhanden sind. Bei der Überprüfung von verschachtelten Klammerausdrücken liegt ein sehr ähnliches Problem vor, denn die Anzahl der öffnenden Klammern muss gleich der Anzahl der schließenden Klammern sein.

Endliche Automaten können Sprachen wie (4.5) nicht erkennen, da sie durch ihre Zustände nur endlich viele Informationen spreichern können. Es muss jedoch eine beliebige Anzahl von 0en und 1en gespeichert werden können, *i.e.* also im Allgemeinen eine *unbeschränkte* Anzahl. Denn um festzustellen, dass ein Wort die Form $\mathtt{0}^n\mathtt{1}^n$ hat, muss der endliche Automat n Zustände enthalten (n unbeschränkt), die die 0en zählen. Dann lässt sich anschließend feststellen, ob die gleiche Anzahl an 1en folgt.

Erweitert man die endliche Automaten mit einem zusätzlichen Speicher, der in der Kapazität nicht beschränkt ist, dann ist man in der Lage, eine Sprache der Form (4.5) zu erkennen. Diese Erweiterung endlicher Automaten haben wir bereits in den Abbildungen [4.8] und [4.9] skizziert. Das derart erweiterte Automatenmodell ergibt das Konzept der **Kellerautomaten**.

Abarbeitung von Inputstrings

- Zu Beginn der Verarbeitung befindet sich die Zustandskontrolle[10] in einem Anfangszustand. Der Stackspeicher ist leer mit Ausnahme eines Symbols, das den 'Boden' des Kellerspeichers kennzeichnet. Das zu bearbeitende Wort steht auf dem Eingabeband, beginnend auf dem ersten Feld. Sonst ist das Eingabeband leer. Der Lesevorgang beginnt auf dem ersten Feld des Eingabebandes.

- Der Lesekopf der Maschine geht schrittweise auf dem Eingabenabd nach rechts und liest die Zeichen, die in den einzelnen Zellen des Inputbandes stehen.

- Für die einzelnen Arbeitsschritte gibt es zwei Möglichkeiten:

  (a) Abhängig vom momentanen Zustand, dem gelesenen Zeichen auf dem Eingabeband und dem obersten Zeichen auf dem Stack geht die Steuereinheit in einen Folgezustand über, schreibt ein *Wort* auf den Stack und rückt den Lesekopf auf dem Eingabeband eine Zelle weiter.

[10] Wir nennen diese Funktionseinheit auch einfach Steuereinheit.

(b) Abhängig vom momentanen Zustand und dem obersten Zeichen auf dem Stack geht die Steuereinheit in den Folgezustand über. Die Steuereinheit schreibt ein Wort in den Stack, rückt aber den Lesekopf auf dem Eingabeband nicht weiter. Dabei spielt das Zeichen auf dem Eingabeband keine Rolle.

- Das Eingabewort wird von dem Pushdown–Automaten erkannt, wenn er sich nach dem vollständigen Lesen des Wortes entweder in einem Acceptzustand befindet (bei Kellerautomaten mit Acceptzuständen) oder wenn der Stack leer ist[11] (bei Kellerautomaten ohne Acceptzustände).

[11] bis auf den Stackbegrenzer

4.6 Nichtdeterministische Pushdown–Automaten

Wir betrachten hier nur Kellerautomaten mit Acceptzuständen, da sie die direkte Verallgemeinerung der endlichen Automaten darstellen. Wir betrachten nichtdeterministische Pushdown–Automaten, da im Gegensatz zu den endlichen Automaten, die nichtdeterministsichen Kellerautomaten mächtiger sind als die deterministischen Kellerautomaten. Wir werden sehen, dass nur mit Hilfe des Nichtdeterminismus alle kontextfreien Sprachen erkannt werden können.

Definition [4.24]:

Ein **nichtdeterministischer Kellerautomat** oder **nichtdeterministischer Pushdown–Automat** mit Endzuständen ist ein 7–Tupel

$$M = (Q, \Sigma, \Gamma, \delta, q^s, \bot, F)$$

mit:

Q**:** Endliche Menge, Menge der Zustände.

Σ**:** Endliche Menge, das Bandalphabet.

Γ**:** Endliche Menge, das Stackalphabet, $\Sigma \cap \Gamma = \emptyset$.

δ**:** Die Überführungsrelation

$$\delta \ : \ Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \longrightarrow Q \times \Gamma^*.$$

Die Elemente aus δ nennt man **Anweisungen**.

q^s**:** ist der Anfangszustand, $q^s \in Q$.

$\bot$**:** ist der Stackbegrenzer, $\bot \in \Gamma$,

F**:** F ist die Menge der Acceptzustände, $F \subseteq Q$.

Anmerkung:

Die Anweisungen sind Relationen der Form

$$(q, a, A, p, \alpha) \in \delta,$$

die wir in der Regel in der Form schreiben:

$$qaA \longmapsto p\alpha.$$

Die letztere Schreibweise trennt den Bedingungsteil qaA von dem Aktionsteil $p\alpha$.

Eine Anweisung $qaA \longmapsto p\alpha$ hat die folgende Bedeutung, wobei zwei Fälle zu unterscheiden sind:

$a \in \Sigma$ Liest die Maschine M im Zustand $q \in Q$ das Eingabezeichen $a \in \Sigma$, und ist $A \in \Gamma$ das oberste Stacksymbol, dann kann M in den Zustand $p \in Q$ übergehen und das Wort $\alpha \in \Gamma^*$ auf den Stack legen. Der Lesekopf auf dem Eingabenad rückt eine Zelle nach rechts.

$a = \lambda$ Dies nennt man einen λ–Übergang oder spontaner Übergang.
Ist die Maschine M im Zustand $q \in Q$ und ist $A \in \Gamma$ das oberste Zeichen auf dem Stack, dann kann M ohne etwas zu lesen in den Zustand p übergehen und A durch das Wort $\alpha \in \Gamma^*$ ersetzen. Hierbei wird der Lesekopf auf dem Eingabeband nicht weitergerückt.

Bei der Ersetzung auf dem Stack gilt die folgende Reihenfolge: Ist $\alpha = B_1 B_2 \dots B_n$, so ist nach der Ersetzung das Zeichen $B_1 \in \Gamma$ das oberste Stacksymbol.

Anmerkung:

1. Das Wort *kann* wird in den obigen Formulierungen aus dem Grund verwendet, weil eine Anweisung aufgrund des Nichtdeterminismus nur eine unter eventuell mehreren Möglichkeiten darstellt.

2. Eine Anweisung der Form
$$qaA \longmapsto p\lambda$$
entspricht einer pop–Operation, da bei jedem Schritt grundsätzlich das oberste Stacksymbol entfernt wird.

3. Eine Anweisung der Form
$$qaA \longmapsto pBA$$
ist eine push–Operation, durch die das Zeichen B auf den Stack gesetzt wird. Das zunächst entfernte Zeichen A wird wieder auf den Stack gesetzt, bevor das Zeichen B darübergelegt wird.

Ein nichtdeterministischer Kellerautomat stoppt, falls er für eine Konfiguration, die durch den Zustand $q \in Q$, das Inputsymbol $a \in \Sigma$ und das Stacksymbol $A \in \Gamma$ gegeben ist, keine Anweisung erhält. Ein besonderer Fall liegt vor, wenn der Stack leer ist. Dann gibt es per Definition keine Anweisung, der Pushdown–Auotmat stoppt ebenfalls.

Mit Hilfe von spontanen Übergängen können Kellerautomaten — im Gegensatz zu endlichen Automaten — endlos weiterlaufen ohne zu terminieren. Ein solches Szenario wird zum Beispiel durch den Übergang $q\lambda K \longmapsto qK$ verursacht, dieser Übergang ist allerdings sinnlos.

Ein Inputwort muss vollständig gelesen werden, damit es erkannt wird, und der Pushdown Automat muss anschließend in einem Acceptzustand terminieren.

Definition [4.25]:

Sei

$$M = (Q, \Sigma, \Gamma, \delta, q^s, \bot, F)$$

ein Kellerautomat. Das Wort $w \in \Sigma^*$ liegt auf dem Eingabeband, beginnend mit der ersten Zelle. Der Automat startet mit dem Lesekopf auf dem ersten Feld im Startzustand, der Stackspeicher ist leer bis auf den Stackbegrenzer.

Das Wort w wird von M erkannt (akzeptiert), wenn es in M eine Möglichkeit gibt, w vollständig zu lesen und der Automat terminiert in einem Acceptzustand.

Die von M erkannte Sprache ist die Menge

$$L(M) = \{w \in \Sigma^* \mid w \text{ wird von } M \text{ erkannt}\}.$$

Beispiel [4.66]

Wir wollen einen Pushdown–Automaten M angeben, der die Sprache

$$L = \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid w = \mathtt{0}^n\mathtt{1}^n, n \in \mathbb{N}, n \geq 1\}$$

akzeptiert. Da der Stackspeicher nicht begrenzt ist, kann eine beliebige Anzahl n von Zeichenpaaren verglichen werden.

Zweckmäßig ist, sich zunächst zu überlegen, wie der Automat arbeitet, ohne in den Formalismus zu verfallen. Der Automat muss feststellen, ob gleich viele `0`en und `1`en in dem Inputstring vorkommen. Dies kann man dadurch realisisen, dass der Automat für jede gelesene `0` einen Merker auf den Stack legt. Wenn die erste `1` gelesen wird, dann wir ein Zeichen vom Stack gepoppt. Wenn der String vollständig gelesn ist und der Stack bis auf den Begrenzer leer ist, wird der String akzeptiert. Voarusgesetzt ist, dass die `0`en und `1`en auf dem Inputband in der richtigen Reihenfolge stehen.

Wir setzen

$$M = (Q, \Sigma, \Gamma, \delta, q^s, \bot, F)$$

mit

- den drei Zuständen $Q = \{q_0, q_1, q_2\}$,
- dem Bandalphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$,
- dem Stackalphabet $\Gamma = \{\bot, A\}$,
- dem Startzustand q_0,
- dem Stackbegrenzer $\bot \in \Gamma$,

- und der Übergangsrelation δ. Diese kann man auf verschiedene Weise angeben:

$$\delta(q_0, \mathtt{0}, \bot) = \{(q_0, A\bot)\} \tag{4.6a}$$
$$\delta(q_0, \mathtt{0}, A) = \{(q_0, AA)\} \tag{4.6b}$$
$$\delta(q_0, \mathtt{1}, A) = \{(q_1, \lambda)\} \tag{4.6c}$$
$$\delta(q_1, \mathtt{1}, A) = \{(q_1, \lambda)\} \tag{4.6d}$$
$$\delta(q_1, \lambda, \bot) = \{(q_2, \bot)\} \tag{4.6e}$$

Tabellarisch kann man diese Relation folgendermaßen angeben:

Input	0	0	1	1	λ	λ
Stack	$\bot$	A	$\bot$	A	$\bot$	A
q_0	$(q_0, A\bot)$	(q_0, AA)		$(q_1\lambda)$		
q_1				$(q_1\lambda)$	$(q_2, \bot)$	
q_2						

Sehen wir uns anhand des Strings `000111` an, wie diese Maschine arbeitet.

Step 1: Lesen der ersten `0`
Die Maschine befindet sich im Startzustand q_0, der Stackinhalt ist $\bot$ und es wird das Eingabesymbol `0` gelesen.[12] Wegen des Übergangs (4.6a) bleibt die Maschine im Zustand q_0, nimmt das Symbol $\bot$ vom Stack und setzt dann den String $A\bot$ auf den Stack, *i.e.* effektiv wird also für die gelesene `0` ein 'Merker' A auf dem Stack abgelegt.

Step 2: Lesen der zweiten `0`
Die Maschine ist im Zustand q_0, der oberste Stackinhalt ist ein A und sie liest vom Inputband eine `0`. Gemäß der Übergangsfunktion (4.6b) bleibt die Maschine im Zustand q_0, nimmt das Stackzeichen A vom Stack und legt den String `AA` auf dem Stack ab.

Step 3: Lesen der dritten `0`
Die Maschine ist im Zustand q_0, der oberste Stackinhalt ist ein A und das Inputbandzeichen ist eine `0`. Gemäß der Übergangsfunktion (4.6b) bleibt die Maschine wieder im Zustand q_0, nimmt das A vom Stack und legt den String `AA` auf dem Stack ab.

Step 4: Lesen der ersten `1`
Die Maschine ist im Zustand q_0, das oberste Stacksymbol ist `A` und es wird

[12] Dies ist eine Möglichkeit, das unterste Element des Stacks zu charakterisieren, *i.e.* die Startkonfiguration des Automaten besteht also aus dem Startzustand q_s und der Stackinhalt ist $\bot$. Alternativ kann natürlich die Startkonfiguration aus einem Startzustand q_s und einen komplett leeren Stack bestehen. Der erste Übergang ist dann immer ein nicht–deterministischer λ–Übergang — dann wird also kein Inputsymbol verarbeitet — und die Maschine setzt ein Kennzeichner (*e.g.* $\bot$) auf den Stack.

vom Inputband die erste `1` gelesen. Gemäß der Übergangsfunktion (4.6c) geht die Maschine in der Zustand q_1 über, nimmt das oberste Zeichen `A` vom Stack und legt nichts auf dem Stack ab.

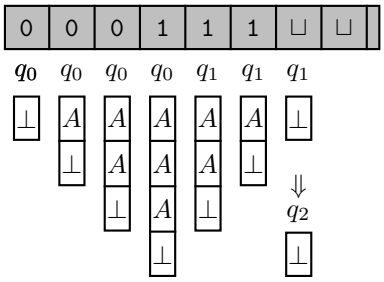
Step 5: Lesen der zweiten `1`
Die Maschine ist im Zustand q_1, das Stacksymbol ist ein `A` und das Inputzeichen ist die nächste `1`. Gemäß der Übergangsfunktion bleibt die Maschine im Zustand q_1 und nimmt ein `A` vom Stack weg.

Step 6: Lesen der dritten `1`
Die Maschine ist im Zustand q_1 und liest vom Inputband die dritte `1` und auf den Stack liegt ein `A`. Gemäß der Übergangsfunktion bleibt die Maschine im Zustand q_1 und nimmt das letzte `A` vom Stack weg.

Step 7: Kein Eingabesymbol mehr vorhanden
Die Maschine ist im Zustand q_1, der Stackinhalt ist der Begrenzer $\perp$ und es ist kein Inputsymbol mehr zu lesen. Nach Gl. (4.6e) geht die Maschine in den Zustand q_2 über, lässt den Begrenzer $\perp$ auf dem Stack. Nach der Definition der Akzeptanz durch einen Acceptzustand, wird der String `000111` daher vom Automaten M akzeptiert.

Die Folge von Zuständen, die der Automat P_1 bei der Verarbeitung des Strings `000111` durchläuft ist also:

$$
\begin{aligned}
(q_0, \texttt{000111}, \perp) &\longrightarrow (q_0, \texttt{00111}, \texttt{A}\perp) \\
&\longrightarrow (q_0, \texttt{0111}, \texttt{AA}\perp) \\
&\longrightarrow (q_0, \texttt{111}, \texttt{AAA}\perp) \\
&\longrightarrow (q_1, \texttt{11}, \texttt{AA}\perp) \\
&\longrightarrow (q_1, \texttt{1}, \texttt{A}\perp) \\
&\longrightarrow (q_1, \lambda, \perp) \\
&\longrightarrow (q_2, \lambda, \perp) \quad \longrightarrow \quad \textsc{Stop}.
\end{aligned}
$$

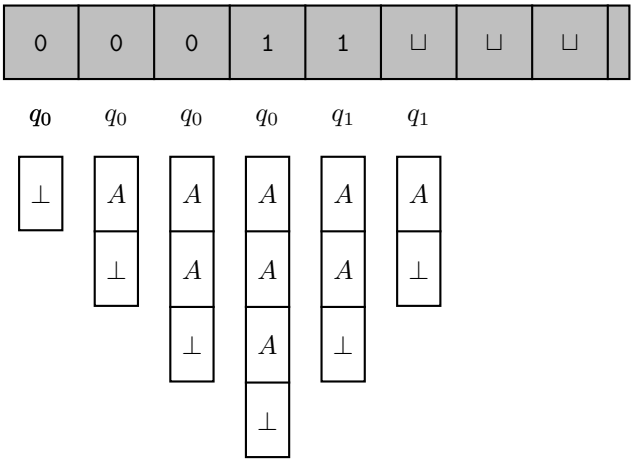


Die Strings

$$w_1 = \texttt{00011},$$
$$w_2 = \texttt{011},$$
$$w_3 = \texttt{0101}$$

werden nicht akzeptiert.

(a) Das Wort $w_1 = \texttt{00011}$ wird nicht akzeptiert, weil sich der Automat nicht in einem Acceptzustand befindet, wenn der String komplett gelesen wurde.



(b) Die Maschine terminiert nicht in einem Acceoptzustand. Es werden die folgenden Konfigurationen durchlaufen:

$$\begin{aligned}(q_0, \texttt{011}, \bot) &\overset{(4.6a)}{\Longrightarrow} (q_0, \texttt{11}, A\bot) \\ &\overset{(4.6b)}{\Longrightarrow} (q_1, \texttt{1}, \bot)\end{aligned}$$

An dieser Stelle terminiert die Verarbeitung, weil es für die aktuelle Konfiguration keinen Folgezustand gibt. Der Zustand q_1 ist kein Acceptzustand, daher wird der String nicht akzeptiert.

(c) Bei der Verarbeitung des Wortes $w_3 = \texttt{0101}$ werden folgende Schritte durchlaufen:

$$\begin{aligned}(q_0, \texttt{0101}, \bot) &\overset{(4.6a)}{\Longrightarrow} (q_0, \texttt{101}, A\bot) \\ &\overset{(4.6c)}{\Longrightarrow} (q_1, \texttt{01}, \bot)\end{aligned}$$

Es gibt hier ebenfalls keinen Folgezustand, daher terminiert die Verarbeitung. Der Automat befindet sich nicht in einem Acceptzustand, daher wird der String nicht akzeptiert.

Der Kellerautomat des Beispiels [4.66] ist ein deterministischer Kellerautomat, denn bei jeder Konfiguration gibt es höchstens eine Möglichkeit für die Weiterverarbeitung. Das folgende Beispiel zeigt einen nichtdeterministischen Kellerautomaten.

Beispiel [4.67]

Gegeben ist das Alphabet $\Sigma = \{0, 1\}$. Wir untersuchen die Sprache

$$L = \{w \in \{0,1\}^* \mid w = x \circ x^R, x \in \Sigma^*\}, \tag{4.7}$$

wobei x^R das Spiegelbild des Strings x ist, *i.e.* wenn

$$x = a_1 a_2 \dots a_n, a_i \in \Sigma, i = 1, \dots n,$$

dann ist der gespiegelte String:

$$x = a_n a_{n-1} \dots a_2, a_1, a_i \in \Sigma, i = 1, \dots n.$$

Die Zeichenketten der Form (4.7) nennt man **Palindrome**, daher nennt man die Sprache (4.7) die Sprache der Palindrome über dem Alphabet Σ. Beispielsweise sind

10011 11001 und 000 000

Palindrome, die Strings

00100 oder 11111

jedoch nicht.

Im diesem Beispiel[13] geben wir einen Pushdown Automaten an, der die Sprache (4.7) erkennt. Die informelle Beschreibung des Pushdown Automaten, der diese Sprache erkennt lautet folgendermaßen:

> Wenn die Symbole des Inputstrings gelesen werden, legt der Automat die gelesenen Symbole auf dem Stack der Reihe nach ab. Also, liest der Automat eine 0, wird ein A auf den Stack gelegt, liest er eine 1, wird ein B auf den Stack abgelegt. Das Problem ist die Mitte des Wortes d zu finden. Bei jedem Schritt wird nichtdeterministisch vermutet, dass die Mitte des Strings erreicht ist. Dann beginnt der Automat damit, für jedes gelesenen Symbol auf dem Inputband ein Symbol vom Stack zu nehmen. Dabei wird verglichen, ob diese beiden Symbole gleich sind. Stellt der Automat fest, dass bei dem Vergleich jedesmal eine Übereinstimmung der Symbole vom Stack und Inputstring vorliegt *und* der Stack leer ist, wenn der Inputstring zu Ende ist, dann ist der String zu akzeptieren, andernfalls nicht.

[13] Siehe [33], Chap. 6.1.

Formal beschreiben wir den Automaten durch das folgende 7–Tupel

$$M = \big(Q, \Sigma, \Gamma, \delta, q^s, \bot, F\big),$$

mit

(a) Der Menge der Zustände

$$Q = \big\{q_1, q_2, q_3\big\}.$$

(b) Dem Inputalphabet

$$\Sigma = \big\{\mathtt{0}, \mathtt{1}\big\}.$$

(c) Dem Stackalphabet

$$\Gamma = \big\{A, B, \bot\big\}$$

(d) Dem Startzustand $q^s = q_1$.

(e) Dem Stackbegrenzer $\bot$

(f) Der Menge der Acceptzustände $F = \{q_3\}$.

(g) Der Übergangsrelation δ, mit

$$\begin{aligned}
\delta(q_1, 0, \bot) &= \{(q_1, A\bot)\}, && (4.8a)\\
\delta(q_1, 1, \bot) &= \{(q_1, B\bot)\}, && (4.8b)\\
\delta(q_1, 0, A) &= \{(q_1, AA)\}, && (4.8c)\\
\delta(q_1, 0, B) &= \{(q_1, AB)\}, && (4.8d)\\
\delta(q_1, 1, A) &= \{(q_1, BA)\}, && (4.8e)\\
\delta(q_1, 1, B) &= \{(q_1, BB)\}, && (4.8f)\\
\delta(q_1, \lambda, \bot) &= \{(q_2, \bot)\}, && (4.8g)\\
\delta(q_1, \lambda, A) &= \{(q_2, A)\}, && (4.8h)\\
\delta(q_1, \lambda, B) &= \{(q_2, B)\}, && (4.8i)\\
\delta(q_2, 0, A) &= \{(q_2, \lambda)\}, && (4.8j)\\
\delta(q_2, 1, B) &= \{(q_2, \lambda)\}, && (4.8k)\\
\delta(q_2, \lambda, \bot) &= \{(q_3, \bot)\}. && (4.8l)
\end{aligned}$$

❶ Eine der beiden ersten Regeln (4.8a) und (4.8b) wird zu Beginn der Abarbeitung des Strings angewendet. Wird eine `0` gelesen, dann wird ein `A` auf den Stack gesetzt, wird eine `1` gelesen, dann ein `B`. Dadurch wird das erste Zeichen gelesen und verarbeitet, der Stackbegrenzer bleibt, um das untere Ende des Stacks zu markieren.

❷ Die vier Regeln

$$\begin{aligned}\delta(q_1, 0, A) &= \{(q_1, AA)\},\\ \delta(q_1, 0, B) &= \{(q_1, AB)\},\\ \delta(q_1, 1, A) &= \{(q_1, BA)\},\\ \delta(q_1, 1, B) &= \{(q_1, BB)\},\end{aligned}$$

lassen den Kellerautomaten im Zustand q_1 und setzen für jedes gelesene Inputsymbol den entsprechenden Marker auf den Stack, wobei das oberste Stackelement unverändert bleibt.

❸ Durch die drei Regeln

$$\begin{aligned}\delta(q_1, \lambda, \perp) &= \{(q_2, \perp)\},\\ \delta(q_1, \lambda, A) &= \{(q_2, A)\}\\ \delta(q_1, \lambda, B) &= \{(q_2, B)\}\end{aligned}$$

verzweigt der PDA nicht–deterministisch vom Zustand q_1 in den Zustand q_2 ohne den Stackinhalt zu ändern.

❹ Ist der PDA nun im Zustand q_2, können Inputsymbole mit Stacksymbolen verglichen werden, bei Übereinstimmung wird das Stacksymbol gepopt. Dies erzielt man durch die Übergänge

$$\begin{aligned}\delta(q_2, 0, A) &= \{(q_2, \lambda)\},\\ \delta(q_2, 1, B) &= \{(q_2, \lambda)\}.\end{aligned}$$

❺ Ist der Stackbegrenzer dann das oberste Stackelement und die Maschine ist im Zustand q_2, dann müssen wir einen Input der Form ww^R verarbeitet haben, der Automat geht in den Zustand q_3 über und akzeptiert den String.

Wir betrachten die Verarbeitung des Strings

$$w = \texttt{0110}$$

durch diesen Pushdown–Automaten.

Die Abbildung [4.10] zeigt die Konfigurationen, die der Kellerautomat annimmt, wenn der String `0110` verarbeitet wird.

Step 1: Der Automat ist Startzustand q_1, liest das Inputzeichen `0`, der Stack hat das Zeichen $\perp$. Wegen Gl. (4.8a) haben wir den Übergang

$$(q_1, \texttt{0110}, \perp) \Longrightarrow (q_1, \texttt{110}, A\perp),$$

wegen Gl. (4.8g) verzweigt der Automat im Startzustand spontan gemäß:

$$(q_1, \texttt{0110}, \perp) \Longrightarrow (q_2, \texttt{0110}, \perp).$$

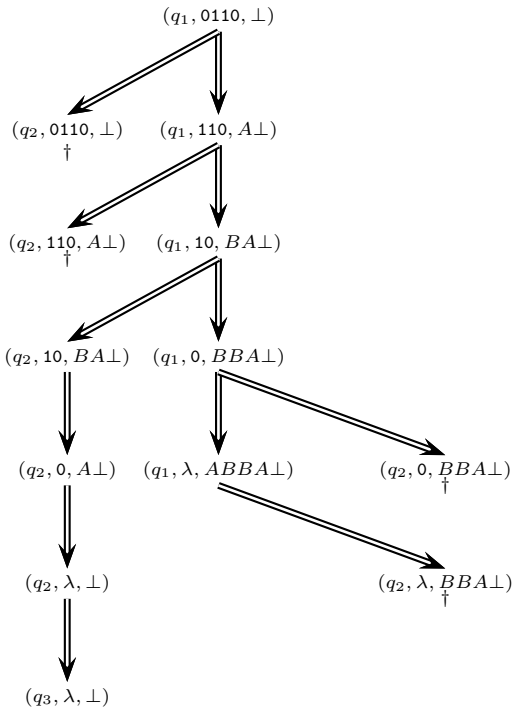


Abbildung 4.10: Konfigurationen des Automaten bie der Verarbeitung des Strings $w = 0110$.

Dieser Verarbeitungszweig terminiert, da die Konfiguration $(q_2, \mathtt{0}, \bot)$ keine Nachfolgekonfiguration hat.

Step 2: Der Automat ist q_1, liest das Inputzeichen `1`, der Stack hat das oberste Zeichen A. Wegen Gl. (4.8e) haben wir den Übergang

$$(q_1, \mathtt{110}, A\bot) \Longrightarrow (q_1, \mathtt{10}, BA\bot),$$

gleichzeitig verzweigt der Automat nichtdeterministisch durch den Übergang:

$$(q_1, \mathtt{110}, A\bot) \Longrightarrow (q_2, \mathtt{110}, A\bot).$$

Dieser Zweig stirbt ab, da es keinen Folgezustand gibt.

Step 3: Der Automat ist im Zustand q_1, er liest das Inputzeichen `1`, der Stack hat das oberste Zeichen B. Wegen Gl. (4.8f) geht der Automat über in:

$$(q_1, \mathtt{10}, BA\bot) \Longrightarrow (q_1, \mathtt{0}, BBA\bot).$$

Die nichtdeterministische Verzweigung liefert hier;

$$(q_1, \mathtt{10}, A\bot) \Longrightarrow (q_2, \mathtt{10}, BA\bot).$$

Dieser Zweig stirbt nicht ab, da der Übergang (4.8k) auf einen nachfolgenden Zustand führt. Damit wird jetzt der Stack Zeichen für Zeichen abgebaut. Man erkennt, dass durch diesen Mechanismus des Nichtdeterminismus dieser Zweig im in die Konfiguration $(q_2, \lambda, \bot)$ übergeht, was schließlich durch den Übergang (4.8l) in der Acceptkonfiguration $(q_3, \lambda, \bot)$ terminiert. Dadurch wird der String akzeptiert.

$\Longrightarrow$ Übungen [4.43] und [4.44].

Die Sprache (4.7) der Palindrome über einem Alphabet ist kontextfrei. Eine Grammatik, die diese Sprache generiert, ist

$$G = (N, T, P, S)$$

mit $N = \{S, X\}, T = \{\mathtt{0}, \mathtt{1}\}$, Startvariable S und den Produktionen P:

$$\begin{array}{lcl} S & \longrightarrow & \lambda \\ S & \longrightarrow & X \\ X & \longrightarrow & \mathtt{0}X\mathtt{1} \\ X & \longrightarrow & \mathtt{1}X\mathtt{0} \\ X & \longrightarrow & \mathtt{00} \\ X & \longrightarrow & \mathtt{11} \end{array}$$

Es ist offensichtlich, dass die Mitte eines Wortes $w \in L$ durch einen Kellerautomaten nicht deterministsich festgestellt werden kann. Betrachtet man beispielsweise die beiden Strings `0000` und `00000`, dann hat der Pushdown Automat keine Möglichkeit, nach dem Lesen der ersten beiden Nullen `00` vorauszusehen, dass er im ersten Wort zum Löschen übergehen muss, dagegen im zweiten Wort noch eine weitere `0` abspeichern muss.

Die Sprache `PALINDROM` ist der Prototy einer kontextfreien Sprache, die *nicht* durch einen deterministischen Kellerautomaten erkannt werden kann. Dies impliziert, dass nichtdeterministische Kellerautomaten mächtiger sind als deterministsiche Kellerautomaten. Bei den endlichen Automaten sind beide Konzepte äquivalent.

4.7 Kontextfreie Sprachen und Kellerautomaten

Die nichtdeterministischen Kellerautomaten erkennen genau die kontextfreien Sprachen. Dies ist die Aussage des folgenden Satzes.

Theorem [4.18]:

Sei L eine Sprache. dann sind die folgenden Aussagen äquivalent.

(a) L ist eine kontextfreie Sprache.

(b) Es existiert ein nichtdeterministische Kellerautomat P, der L erkennt.

Beweis:

Wir wollen hier nur die für die Entwicklung von Parsern relevante Richtung $(a) \Longrightarrow (b)$ zeigen.[14]

(a)$\Longrightarrow$ (b) Konstruktion eines nichtdeterministischen Kellerautomaten aus einer kontextfreien Grammatik.

Gegeben ist eine kontextfreie Grammatik

$$G = (N, T, P, S),$$

mit $L = L(G)$. Wir definieren den folgenden Kellerautomaten

$$P = (Q, \Sigma, \Gamma, \delta, q^s, \bot, F)$$

mit

$$\begin{aligned} Q &= \{q_1, q_2, q_3\}, \\ \Sigma &= T, \\ \Gamma &= N \cup T \cup \{\bot\}, \\ q^s &= q_1 \\ \bot &= \text{ Stackbegrenzer} \\ F &= \{q_3\} \end{aligned}$$

und der Übergangsrelation δ:

$$\begin{array}{rcll} q_1\lambda\bot & \longmapsto & q_2S\bot & \\ q_2\lambda A & \longmapsto & q_2\alpha & \forall\, A \longrightarrow \alpha \in P \\ q_2aa & \longmapsto & q_2\lambda & \forall a \in T, \\ q_2\lambda\bot & \longmapsto & q_3\bot & \end{array}$$

[14] Einen vollständigen Beweis, der auch die Gegenrichtung zeigt, findet man in dem Buch von Michael Sipser [44], chapter 2, pp. 106.

Es gilt $L(P) = L(G)$, denn der so konstruierte Kellerautomat ist exakt der Erkennungsalgorithmus aus Abschnitt [4.4]. Nach der erfolgreichen Linksableitung ist der Stack leer bis auf den Stackbegrenzer $\perp$. Mit der letzten Anweisung geht der Automat P in einen Acceptzustand über.

Beispiel [4.68]

Wir betrachten die kontextfreie Grammatik G, die die Sprache

$$L = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid \mathtt{w} = \mathtt{0}^{\mathtt{n}}\mathtt{1}^{\mathtt{n}}, \mathtt{n} \geq \mathtt{1}\}$$

generiert, *i.e.*

$$G = (N, T, P, S)$$

mit der Variablenmenge, Alphabet T, Produktionen P und Startvariable S:

$$\begin{aligned} N &= \{S\}, \\ T &= \{\mathtt{0},\mathtt{1}\}, \\ P &= \{S \longrightarrow \mathtt{0}S\mathtt{1}, S \longrightarrow \mathtt{01}\}, \\ S &= S. \end{aligned}$$

Nach dem obigen Verfahren definieren wir einen Pushdown–Automaten

$$P = (Q, \Sigma, \Gamma, \delta, q^s, \perp, F)$$

mit

$$\begin{aligned} Q &= \{q_1, q_2, q_3\}, \\ \Sigma &= \{\mathtt{0},\mathtt{1}\}, \\ \Gamma &= \{S, \mathtt{0}, \mathtt{1}, \perp\}, \\ q^s &= q_1 \\ \perp &= \perp \qquad\qquad \text{(Stackbegrenzer)}, \\ F &= \{q_3\}, \end{aligned}$$

und den Übergängen

$$\delta(q_1, \lambda, \perp) = (q_2, S\perp) \tag{4.9a}$$
$$\delta(q_2, \lambda, S) = (q_2, \mathtt{0}S\mathtt{1}) \tag{4.9b}$$
$$\delta(q_2, \lambda, S) = (q_2, \mathtt{01}) \tag{4.9c}$$
$$\delta(q_2, \mathtt{0}, \mathtt{0}) = (q_2, \lambda) \tag{4.9d}$$
$$\delta(q_2, \mathtt{1}, \mathtt{1}) = (q_2, \lambda) \tag{4.9e}$$
$$\delta(q_2, \lambda, \perp) = (q_3, \perp). \tag{4.9f}$$

Sehen wir uns an, wie dieser Automat das Wort $w = \texttt{000111}$ verarbeitet.

$$\begin{aligned}
(q_1, \texttt{000111}, \bot) &\overset{(4.9a)}{\Longrightarrow} (q_2, \texttt{000111}, S\bot) \\
&\overset{(4.9b)}{\Longrightarrow} (q_2, \texttt{000111}, \texttt{0}S\texttt{1}\bot) \\
&\overset{(4.9d)}{\Longrightarrow} (q_2, \texttt{00111}, S\texttt{1}\bot) \\
&\overset{(4.9b)}{\Longrightarrow} (q_2, \texttt{00111}, \texttt{0}S\texttt{11}\bot) \\
&\overset{(4.9d)}{\Longrightarrow} (q_2, \texttt{0111}, S\texttt{11}\bot) \\
&\overset{(4.9c)}{\Longrightarrow} (q_2, \texttt{0111}, \texttt{0111}\bot) \\
&\overset{(4.9d)}{\Longrightarrow} (q_2, \texttt{111}, \texttt{111}\bot) \\
&\overset{(4.9e)}{\Longrightarrow} (q_2, \texttt{11}, \texttt{11}\bot) \\
&\overset{(4.9e)}{\Longrightarrow} (q_2, \texttt{1}, \texttt{1}\bot) \\
&\overset{(4.9e)}{\Longrightarrow} (q_2, \lambda, \bot) \\
&\overset{(4.9f)}{\Longrightarrow} (q_3, \lambda, \bot).
\end{aligned}$$

$\Longrightarrow$ Übung [4.48]

4.8 Übungen

Übung 4.42:

Geben Sie eine kontextfreie Grammatik G an, für die gilt: $L = L(G)$ mit

$$L = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w = \mathtt{0}^i\mathtt{1}^j, i < j, i, j \in \mathbb{N}_0\}.$$

Definieren Sie die Grammatik vollständig.

Übung 4.43:

Sei M der Pushdown Automat, definiert durch

$Q = \{q_1, q_2, q_3, q_4\}$,
$\Sigma = \{a, b\}$,
$\Gamma = \{A, \bot\}$,
$q^s = q_1$
$F = \{q_3, q_4\}$.

und

$$\begin{aligned}
\delta(q_1, a, \bot) &= \{(q_1, A\bot)\}\\
\delta(q_1, a, A) &= \{(q_1, AA)\}\\
\delta(q_1, b, A) &= \{(q_2, \lambda)\}\\
\delta(q_1, \lambda, A) &= \{(q_4, A)\}\\
\delta(q_1, \lambda, \bot) &= \{(q_4, \bot)\}\\
\delta(q_2, b, A) &= \{(q_2, \lambda)\}\\
\delta(q_2, \lambda, A) &= \{(q_3, A)\}.\\
\delta(q_2, \lambda, \bot) &= \{(q_3, \bot)\}.\\
\delta(q_4, a, \bot) &= \{(q_4, \bot)\}.
\end{aligned}$$

(a) Man beschreibe die Sprache L, die von M akzeptiert wird.

(b) Man erstelle die Ableitungen der Strings `aab`, `abb` und `aba` in M.

(c) Man zeige, dass $\mathtt{aabb}, \mathtt{aaab} \in L(M)$.

Übung 4.44:

Man erstelle Pushdown Automaten, die die folgenden Sprachen akzeptieren.

(a) Die Sprache über dem Alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, deren Worte die gleiche Anzahl von as und bs hat.

(b) Die Sprache über dem Alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, deren Worte die doppelte Anzahl von as wie bs hat.

(c) $\{a^m b^n, 0 \leq m \leq n \leq 2m\}$.

Geben Sie jeweils auch informelle Beschreibungen an.

Übung 4.45:

Geben Sie für jede der folgenden Sprachen über dem Alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ bzw. $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{2}\}$ eine kontextfreie Grammatik G an mit $L(G) = L_i, i = 1, \ldots 7$.

(a) $L_1 = \{\mathtt{0}^n\mathtt{1}^m, m \geq n, n \geq 1\}$,

(b) $L_2 = \{\mathtt{0}^n\mathtt{1}^n, n \geq 1\} \cup \{\mathtt{0}^n\mathtt{1}^{2n}, n \geq 1\}$

(c) $L_3 = \{\mathtt{0}^n\mathtt{1}^m\mathtt{0}^{n+m}, m, n \geq 1\}$,

(d) $L_4 = \{\mathtt{0}^{2n}\mathtt{1}^m, n \geq 1, m \geq n\}$,

(e) $L_5 = \{\mathtt{0}^n\mathtt{1}^m, m, n \geq 1 m \neq n\}$,

(f) $L_6 = \{\mathtt{0}^i\mathtt{1}^j\mathtt{2}^k, i + j \geq 1, k \leq i + j\}$,

(g) $L_7 = \{\mathtt{0}^i\mathtt{1}^j\mathtt{2}^k, i, j, k \geq 1, i = 2j$ oder $k = 2j\}$,

Übung 4.46:

Gegeben ist die folgende Sprache

$$L = \{w \in \{a, b\}^* \mid w = (a^2b)^n, n \geq 0, n \in \mathbb{N}\}.$$

(a) Geben Sie zu dieser Sprache einen Kellerautomaten iP an mit $L = L(P)$.

(b) Zeigen Sie, wie dieser Automat das Wort *aabaab* verarbeitet.

(c) Geben Sie zu dieser Sprache einen deterministischen endlichen Automaten M an mit $L = L(M)$.

(d) Zu welcher/welchen CHOMSKY Sprachklasse/n gehört die Sprache L?

Übung 4.47:

Betrachte die Sprache

$$L = \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid w \text{ enthält doppelt soviele } \mathtt{0}\text{en wie } \mathtt{1}\text{en}\}.$$

(a) Geben Sie einen Kellerautomaten an, der L akzeptiert.

(b) Zeigen Sie, wie der Automat das Wort

$$w = \texttt{100011100000}$$

erkennt.

Übung 4.48:

Gegeben ist die kontextfreie Grammatik

$$G = (N, T, P, S)$$

mit

$$\begin{aligned} N &= \{S\} \\ T &= \{a, b\} \\ P &= \{S \longrightarrow aab \mid bbS\}. \end{aligned}$$

(a) Geben Sie einen Pushdown Automaten

$$M = (Q, \Sigma, \Gamma, \delta, q^s, \bot)$$

an, der zur Grammatik G äquivalent ist.

(b) Geben Sie die Konfigurationsübergänge von M für das Testwort $w = bbbbaab$ an.

Kapitel 5

Turing–Maschinen

Der eingeschränkte Speicherzugriff bei Kellerautomaten erlaubt es nicht, Sprachen wie

$$L = \{0^n 1^n 2^n \mid n \geq 1\}$$

zu erkennen. Die Information über die Anzahl der 0en muss auf dem Stack gelöscht werden, wenn sie mit der Anzahl der 1en verglichen werden soll. Daher steht diese Information nicht mehr zur Verfügung, wenn die 2er abgezählt werden.

Die Aufhebung des eingeschränkten Speicherzugriffs führt zu der Verwendung eines Speichers mit wahlfreiem Zugriff, bei dem jede beliebige Stelle direkt bearbeitet werden kann. Dies korrespondiert mit den Hauptspeicherzugriffen oder Festplattenzugriffen eines Computers.

Man erhält das Automatenmodell der **Turing–Maschine**, mit dem man genau die Typ 0 Sprachen erkennen kann.[1] Die TURING Maschine wurde 1936 von dem britischen Logiker und Mathematiker ALAN TURING[2] entwickelt [125]. Eine spezielle Form der TURING–Maschine, bei der nur der Speicherplatz zur Verfügung steht, der durch die Eingabe belegt ist, ist der **linear beschränkte Automat**, der das Erkennungsmodell der kontextsensitiven Sprachen darstellt.

Wir untersuchen zunächst das Konzept der TURING–Maschine und zeigen, wie mit TURING–Maschinen Sprachen erkannt werden können. Da TURING–Maschinen generell die Arbeitsweise von Computern abstrahieren, werden wir ihre Verwendung als Algorithmenmodell zur Berechnung von Funktionen darstellen. Wir sehen uns anschließend mit das wichtigste Anwendungsgebiet von TU-

[1] Die Literatur über TURING Maschinen ist sehr umfangreich. Nichttechnische Betrachtungen über TURING Maschinen findet man in [8], Kapitel 31, [7], [12], [17] oder [20]. Das Buch von PETZOLD [21] gibt eine sehr detaillierte Analyse der Originalarbeit von TURING [125].

[2] Die Standard–Biographie von ALAN TURING ist die Monographie von ANDREW HODGES [14], siehe auch [6].

Abbildung 5.1: Alan Turing.

ring–Maschinen an, dies ist die **Komplexitätstheorie**.[3] Hierbei geht es um grundsätzliche Fragen im Kontext der Effizienzbetrachtungen.

5.1 Arbeitsweise von Turing–Maschinen

Es gibt viele verschiedene Definitionen einer Turing–Maschine, die jedoch alle äquivelant sind. Für unsere Zwecke ist die folgende Version sinnvoll, die man *einfache* Turing–*Maschine* nennt.

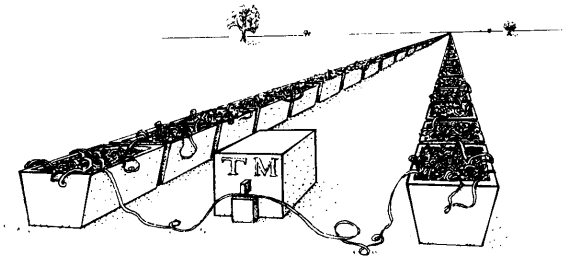
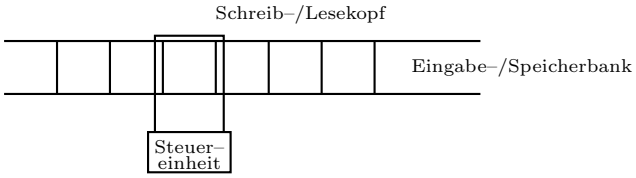


Abbildung 5.2: Eine künstlerische Darstellung einer Turing–Maschine.

Das Eingabeband und der Speicher eines Kellerautomaten legt man zusammen und verwendet ein einziges Speicherband, das sowohl zur Eingabe als auch zur Abspeicherung von Informationen verwendet wird. Das Eingabeband ist beidseitig unbeschränkt und in einzelne Zellen aufgeteilt. Ein Lese–/Schreibkopf kann den Inhalt der einzelnen Zellen lesen und beschriften. Der Schreib–/Lesekopf

[3] Eine sehr lesbare, nicht–technische Einführung in die Thematik der Komplexitätstheorie findet man in [10].

kann sich schrittweise von Zelle zu Zelle auf dem Band nach links oder rechts bewegen.



Verarbeitung von Inputstrings

- Das zu verarbeitende Wort steht auf dem Band, das sonst leer ist. Das Wort wird zeichenweise in die Bandzellen geschrieben. Die TURING–Maschine beginnt an der Zelle, in der das erste Zeichen des Wortes steht, gleichzeitig befindet sich die Steuereinheit der Maschine in einem Anfangszustand. Bei der Eingabe des leeren Wortes λ erfolgt der Start auf einer beliebigen Zelle.

- Bei der schrittweisen Abarbeitung des Wortes wird eine Folge von Aktionen ausgeführt, die vom augenblicklichen Zustand *und* dem gelesenen Zeichen abhängen. Pro Arbeitsschritt wird dabei eine der folgenden Aktionen ausgeführt:

  (R) Bewegung des Schreib–/Lesekopfes um eine Zelle nach rechts,

  (L) Bewegung des Schreib–/Lesekopfes um eine Zelle nach links,

  (t) das gelesene Zeichen wird mit dem Zeichen t überschrieben.

- Das Eingabewort wird durch die TURING–Maschine akzeptiert, wenn sie nach endlich vielen Schritten im (einzigen) Acceptszustand anhält.

Definition [5.26]:

Eine **deterministische Turing–Maschine** TM ist ein 5–Tupel

$$M = (Q, \Sigma, \delta, q^s, q^f)$$

mit:

Q ist eine endliche, nichtleere Menge, die Menge der Zustände

Σ ist eine endliche, nichtleere Menge, das Maschinenalphabet

δ ist die Übergangsfunktion

$$\delta \; : \; \big(Q \setminus \{q^f\}\big) \times (\Sigma \cup \{\sqcup\}) \longrightarrow Q \times (\Sigma \cup \{\sqcup\} \cup \{L, R\}) \,. \quad (5.1)$$

Das Zeichen $\sqcup$ steht für Blank (=Leerzeichen). Beachte gemäß dieser Definition kann die Maschine das Blank lesen und schreiben. L steht für eine Bewegung in Richtung 'Links' und R steht für eine Bewegung des Schreib–/Lesekopfes eine Zelle nach 'Rechts'.

q^s ist der Startzustand, $q^s \in Q$.

q^f ist der Accept– oder Endzustand, $q^f \in Q$.

Anmerkungen:

1. Die Übergangsfunktion δ ist eine endliche Menge von Anweisungen der Form

   $$\delta(q, a) = (q', \alpha),$$

   für die wir auch die folgende Schreibweise verwenden:

   $$q\, a \longmapsto q'\; \alpha.$$

   Diese Schreibweise hat die folgende Bedeutung: Liest die Maschine M im Zustand q das Zeichen a auf dem Eingabeband, dann geht M in den Zustand q' über und führt die Aktion α aus. Gemäss obiger Konvention[4] kann α die Aktion Schreiben, Linksschritt oder Rechtsschritt sein.

[4] Siehe auch die Definition [5.1], die die Form der Übergangsfunkion festlegt. Wir haben die Turing–Maschine so gewählt, dass die Schritte sehr *elementar* sind. Üblicherweise wird die Turing Maschine in der Literatur so definiert, dass die Übergangsfunktion die Form

$$\delta(q, a) = (q', b, R)$$

hat. Dies bedeutet, wenn die TM im Zustand q das Zeichen a auf dem Band liest (linke Seite), dann geht sie in den Zustand q' über, schreibt das Zeichen b in die aktuelle Zelle, *und* geht eine Zelle nach rechts.

2. Das Überschreiben eines Zeichens auf dem Eingabeband, das von dem Blank Zeichen verschieden ist, entspricht dem Löschen des Zeichens.

Eine TURING–Maschine stoppt, wenn sie im Zustand q mit dem Schreib–/Lesekopf auf dem Zeichen a steht und es gibt keine Anweisung für das Paar (q, a). Dies ist gemäß Definition immer der Fall wenn $q = q^f$ ist. Da die Übergangsfunktion δ nicht als total vorausgesetzt wird, kann dies aber auch für andere Paare

$$(q, a) \in (Q \setminus \{q^f\}) \times (\Sigma \cup \{\sqcup\})$$

vorkommen.

Eine TURING–Maschine muss nicht anhalten. Sie kann beispielsweise mit den beiden Anweisungen

$$q_1\, a \longmapsto q_2 R \quad \text{und} \quad q_2\, a \longmapsto q_1 L$$

immer hin– und herspringen, falls sie im Zustand q_1 auf das erste von zwei nebeneinanderstehenden a gerät. Die ist nicht verwunderlich, denn sonst sind Erkennungsprozesse mit TURING–Maschinen immer abbrechende Verfahren, somit Entscheidungsverfahren. Dann gäbe es also aufzählbare Sprachen, die nicht von TURING–Maschinen erkannt werden können.

5.2 Erkennen von Sprachen durch Turing Maschinen

Ein Wort einer Sprache L wird erkannt, wenn die TURING–Maschine in der vorgechriebenen Weise startet und nach endlich vielen Schritten im Endzustand terminiert. Dabei ist es irrelevant, in welcher Weise das Wort gelesen oder verändert wird.

Definition [5.27]:

Sei

$$M = (Q, \Sigma, \delta, q^s, q^f)$$

eine TURING–Maschine. Das Wort $w \in \Sigma^*$ ist auf das sonst leere Eingabeband geschrieben. Der Automat startet mit dem Schreib–/Lesekopf auf dem ersten Zeichen des Eingabewortes im Anfangszustand, bzw. auf einem beliebigen Feld, falls $w = \lambda$. Diese Anordnung heißt **Anfangskonfiguration**.

Das Wort w wird von M erkannt (akzeptiert), falls M ausgehend von der Anfangskonfiguration nach endlich vielen Schritten im Endzustand terminiert.

Die von M erkannte/akzeptierte Sprache ist die Menge

$$L(M) = \{w \in \Sigma^* \mid w \text{ wird von } M \text{ erkannt}\}.$$

Beispiel [5.69]

Wir betrachten das Alphabet $\Sigma = \{a, b, c\}$ und die Sprache $L \subset \Sigma^*$, die aus allen Wörtern besteht, die mit beliebig vielen a beginnen und mit b oder c enden. Dies ist eine reguläre Sprache, denn sie kann durch den regulären Ausdruck

$$\alpha = a^*(b \mid c)$$

beschrieben werden. Wir geben eine TURING–Maschine an, die L erkennt. Diese Maschine ist

$$M = (Q, \Sigma, \delta, q^s, q^f)$$

mit der Menge der Zustände

$$Q = \{q_0, q_1, q_2\},$$

dem Eingabealphabet

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

dem Startzustand q_0, dem Accept–/Endzustand $q^f = q_2$ und den Übergängen δ:

$$\begin{array}{rcl} q_0\, a & \longmapsto & q_0\, R \\ q_0\, b & \longmapsto & q_1\, R \\ q_0\, c & \longmapsto & q_1\, R \\ q_1\, \sqcup & \longmapsto & q_2\, \sqcup \end{array}$$

Die Maschine M verarbeitet das Wort $aaac$ in der folgenden Weise:

$$\begin{array}{rcl} (q_0aaac\sqcup) & \longrightarrow & (aq_0aac\sqcup) \\ & \longrightarrow & (aaq_0ac\sqcup) \\ & \longrightarrow & (aaaq_0c\sqcup) \\ & \longrightarrow & (aaacq_1\sqcup) \\ & \longrightarrow & (aaacq_2\sqcup) \end{array}$$

Man erkennt, dass diese TURING–Maschine wie ein endlicher Automat arbeitet, sie muss nur zusätzlich das Wortende verifizieren. Zu diesem Zweck muss sie nach dem Lesen eines bs oder eines cs noch eine weitere Zelle nach rechts gehen, um festzustellen, dass diese Zelle mit dem $\sqcup$–Zeichen belegt ist und kein weiteres Zeichen des Alphabets mehr folgt. Eine solche Situation liegt vor, wenn die Maschine das Wort $aaabc$ verarbeitet.

$$\begin{array}{rcl} (q_0aaabc\sqcup) & \longrightarrow & (aq_0aabc\sqcup) \\ & \longrightarrow & (aaq_0abc\sqcup) \\ & \longrightarrow & (aaaq_0bc\sqcup) \\ & \longrightarrow & (aaabq_1c\sqcup) \end{array}$$

Die Übergangsfunktion hat keinen Wert für die Konfiguration (q_1, c), daher terminiert die Maschine an diesem Schritt, sie ist jedoch nicht in dem Acceptzustand.

Beispiel [5.70]

Wir geben in diesme Beispiel eine TURING–Maschine an, die die kontextsensitive Sprache

$$L = \{w \in \{\mathtt{0},\mathtt{1},\mathtt{2}\}^* \mid w = \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n, n \geq 1\}$$

erkennt. Informell arbeitet diese Maschine wie folgt:

Step 1: Die Maschine befindet sich im Startzustand und steht auf einer Zelle, die mit `0` beschrieben ist. Die Maschine löscht die gelesene `0` und ersetzt diese durch $\sqcup$.

Step 2: Die `TM` geht schrittweise nach rechts bis zur letzten `1`. Man beachte, dass die letzte `1` dadurch gefunden wird, dass die erste `2` gelesen wird, dann muss die `TM` eine Zelle nach links gesetzt werden.

Step 3: Die letzte `1` wird gelöscht und durch eine `2` überschrieben.

Step 4: Die `TM` geht weiter nach rechts bis zm ersten $\sqcup$ Zeichen.

Step 5: Die `TM` löscht die beiden letzten `2`er.

Step 6: Die Maschine geht nach links bis zum ersten Blankzeichen.

Step 7: Die `TM` geht eine Zelle nach rechts und sitzt wieder auf einer Zelle mit einer `0`.

Step 8: Dann werden die Schritte 1 bis 7 wiederholt.

Auf diese Weise wird ein Wort der Form $\mathtt{0}^n\mathtt{1}^n\mathtt{2}^n$ auf ein Wort $\mathtt{0}^{n-1}\mathtt{1}^{n-1}\mathtt{2}^{n-1}$ reduziert. Der Vorgang wird solange wiederholt, bis alle Zeichen vom Eingabeband gelöscht sind.

Die Maschine ist

$$M = (Q, \Sigma, \delta, q^s, q^f)$$

mit den zehn Zuständen

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9\},$$

dem Inputalphabet $\Sigma = \{a, b, c\}$, dem Startzustand $q^s = q_0$, dem Acceptzustand $q^f = q_9$ und den folgenden Übergängen:

(1)	$q_0\ \mathtt{0}$	$\longmapsto$	$q_1\ \sqcup$	Löschen der `0`
(2)	$q_1\ \sqcup$	$\longmapsto$	$q_2\ R$	eine Zelle nach rechts
(3)	$q_2\ \mathtt{0}$	$\longmapsto$	$q_2\ R$	nach rechts bis erste `1`
(4)	$q_2\ \mathtt{1}$	$\longmapsto$	$q_2\ R$	nach rechts bis erste `2`
(5)	$q_2\ \mathtt{2}$	$\longmapsto$	$q_3\ L$	nach links zur letzten `1`
(6)	$q_3\ \mathtt{1}$	$\longmapsto$	$q_4\ 2$	`1` durch `2` ersetzen
(7)	$q_4\ \mathtt{2}$	$\longmapsto$	$q_4\ R$	nach rechts zum ersten $\sqcup$
(8)	$q_4\ \sqcup$	$\longmapsto$	$q_5\ L$	zurück zur letzten `2`
(9)	$q_5\ \mathtt{2}$	$\longmapsto$	$q_5\ \sqcup$	Löschen der letzten `2`
(10)	$q_5\ \sqcup$	$\longmapsto$	$q_6\ L$	eine Zelle nach links
(11)	$q_6\ \mathtt{2}$	$\longmapsto$	$q_6\ \sqcup$	Löschen der vorletzten `2`
(12)	$q_6\ \sqcup$	$\longmapsto$	$q_7\ L$	Zurück Richtung Wortanfang nach links
(13)	$q_7\ \sqcup$	$\longmapsto$	$q_9\ \sqcup$	alle Zeichen gelöscht
(14)	$q_7\ \mathtt{2}$	$\longmapsto$	$q_8\ L$	nach links
(15)	$q_8\ \mathtt{2}$	$\longmapsto$	$q_8\ L$	nach links
(16)	$q_8\ \mathtt{1}$	$\longmapsto$	$q_8\ L$	nach links
(17)	$q_8\ \mathtt{0}$	$\longmapsto$	$q_8\ L$	nach links
(18)	$q_8\ \sqcup$	$\longmapsto$	$q_0\ R$	Anfang erreicht, wiederhole alle Schritte

Sehen wir uns die Verarbeitung des Strings 001122 durch diese `TM` an.

$$
\begin{aligned}
\sqcup q_0 001122\sqcup &\xrightarrow{(1)} \sqcup q_1 \sqcup 01122\sqcup \\
&\xrightarrow{(2)} \sqcup\sqcup q_2 01122\sqcup \\
&\xrightarrow{(3)} \sqcup\sqcup 0 q_2 1122\sqcup \\
&\xrightarrow{(4)} \sqcup\sqcup 01 q_2 122\sqcup \\
&\xrightarrow{(4)} \sqcup\sqcup 011 q_2 22\sqcup \\
&\xrightarrow{(5)} \sqcup\sqcup 01 q_3 122\sqcup \\
&\xrightarrow{(6)} \sqcup\sqcup 01 q_4 222\sqcup \\
&\xrightarrow{(7)} \sqcup\sqcup 012 q_4 22\sqcup \\
&\xrightarrow{(7)} \sqcup\sqcup 0122 q_4 2\sqcup \\
&\xrightarrow{(7)} \sqcup\sqcup 01222 q_4\sqcup \\
&\xrightarrow{(8)} \sqcup\sqcup 0122 q_5 2\sqcup \\
&\xrightarrow{(9)} \sqcup\sqcup 0122 q_5 \sqcup\sqcup \\
&\xrightarrow{(10)} \sqcup\sqcup 012 q_6 2\sqcup\sqcup \\
&\xrightarrow{(11)} \sqcup\sqcup 012 q_6 \sqcup\sqcup\sqcup \\
&\xrightarrow{(12)} \sqcup\sqcup 01 q_7 2\sqcup\sqcup\sqcup \\
&\xrightarrow{(13)} \sqcup\sqcup 0 q_8 12\sqcup\sqcup\sqcup \\
&\xrightarrow{(16)} \sqcup\sqcup q_8 012\sqcup\sqcup\sqcup \\
&\xrightarrow{(17)} \sqcup q_8 \sqcup 012\sqcup\sqcup\sqcup \\
&\xrightarrow{(18)} \sqcup\sqcup q_0 012\sqcup\sqcup\sqcup \\
&\xrightarrow{(1)} \sqcup\sqcup q_1 \sqcup 12\sqcup\sqcup\sqcup \\
&\xrightarrow{(2)} \sqcup\sqcup\sqcup q_2 12\sqcup\sqcup\sqcup \\
&\xrightarrow{(4)} \sqcup\sqcup\sqcup 1 q_2 2\sqcup\sqcup\sqcup \\
&\xrightarrow{(5)} \sqcup\sqcup\sqcup q_3 12\sqcup\sqcup\sqcup \\
&\xrightarrow{(6)} \sqcup\sqcup\sqcup q_4 22\sqcup\sqcup\sqcup \\
&\xrightarrow{(7)} \sqcup\sqcup\sqcup 2 q_4 2\sqcup\sqcup\sqcup \\
&\xrightarrow{(7)} \sqcup\sqcup\sqcup 22 q_4\sqcup\sqcup\sqcup \\
&\xrightarrow{(8)} \sqcup\sqcup\sqcup 2 q_5 2\sqcup\sqcup\sqcup \\
&\xrightarrow{(9)} \sqcup\sqcup\sqcup 2 q_5 \sqcup\sqcup\sqcup\sqcup \\
&\xrightarrow{(10)} \sqcup\sqcup\sqcup q_6 2\sqcup\sqcup\sqcup\sqcup \\
&\xrightarrow{(11)} \sqcup\sqcup\sqcup q_6 \sqcup\sqcup\sqcup\sqcup\sqcup \\
&\xrightarrow{(12)} \sqcup\sqcup q_7 \sqcup\sqcup\sqcup\sqcup\sqcup\sqcup \\
&\xrightarrow{(13)} \sqcup\sqcup q_9 \sqcup\sqcup\sqcup\sqcup\sqcup\sqcup
\end{aligned}
$$

Allgemeine TURING–Maschinen erkennen genau die Typ 0 Sprachen, diese nennt man auch **rekursiv aufzählbare Sprachen**. Die kontextsensitiven Sprachen werden durch nichtdeterministische, linear beschränkte TURING–Maschinen erkannt. Dies sind TURING–Maschinen, die nur den Spreicherplatz benötigen, der durch die Eingabe belegt ist.[5] Wir wollen diese Zusammenhänge hier nicht weiter verfolgen, denn im Gegensatz zu den regulären und kontextfreien Sprachen ergibt die Konstruktion einer TURING–Maschine aus einer Typ0 Grammatik keinen für die Praxis wichtigen Algorithmus.

Die in den beiden Beispielen betrachteten TURING–Masschinen sind linear beschränkte Automaten. Da die zugehörigen Sprachen kontextsensitiv sind, ist dies auch ausreichend. Sprachen, die nicht kontextsensitiv sind findet man in dem Buch von WEGENER [47].

[5] Dies ist in den Beispielen [5.69] und [5.70] der Fall. Die TURING–Maschine, die im Beispiel [5.70] Wörter der Form `000111222` akzeptiert, bewegt den Schreib–Lesekopf nur über die Zellen des Eingabebands, die mit dem Input belegt sind. Es wird kein Speicherplatz benötigt, um Zwischenergebnisse abzulegen.

5.3 Turing–Berechenbarkeit von Funktionen

Computerprogramme sind auf Rechnern ausführbare Algorithmen. Abhängig von der Eingabe, die sich aus einzelnen Eingabeteilen während des Programmablaufs zusammensetzt, wird nach einer endlichen Folge von elementaren Programmschritten eine eventuell ebenfalls aus Einzelteilen bestehende Ausgabe geliefert, oder das Programm gerät in eine Endlosschleife. Etwas abstrakter formuliert: Jedes Computerprogramm realisiert eine Funktion aus der Menge der zulässigen Eingaben in die Menge der möglichen Ausgaben. Diese Funktion kann partiell sein, *i.e.* sie muss nicht für alle Eingaben definiert sein.

Dieser Sachverhalt wird durch die folgende Definition beschrieben. Die Rolle der Algorithmen respektive Programme wird von TURING–Maschinen übernommen. Die Definition [5.28] stellt eine grundlegende mathematische Präzisierung des Begriffs *Berechenbarkeit von Funktionen* dar. Diese erlaubt es, die Begriffe Computerprogramm und TURING–Maschine als verschiedene Ausprägungen des intuitiven Konzepts der **Algorithmen** aufzufassen.

Definition [5.28]:

Eine n–stellige Funktion

$$f \ : \ \left(\Sigma^+\right)^n \longrightarrow \Sigma^*, \quad (n \geq 1)$$

heißt **Turing berechenbar** wenn es eine TURING Maschine M gibt, für die folgendes zutrifft.

Schreibt man auf das Eingabeband die Argumente $x_1, \ldots, x_n$ getrennt durch das Leerzeichen $\sqcup$:

$$\sqcup \sqcup \ x_1 \ \sqcup \ x_2 \sqcup \cdots\cdots \sqcup \ x_n \ \sqcup \sqcup$$

und setzt man die TM M auf das erste Zeichen von x_1 an, so terminiert M nach endlich vielen Schritten im Endzustand und auf dem Eingabeband steht ausschließlich der Funktionswert

$$\sqcup \sqcup \ f(x_1, x_2 \cdots x_n) \ \sqcup \sqcup$$

Die Funktion f kann partiell oder total sein. f partiell bedeutet, dass f nicht für jedes n–Tupel von Wörtern über dem Alphabet Σ definiert sein muss. Dies ist bei totalen Funktionen der Fall. Falls f für ein n–Tupel von Argumenten nicht definiert ist, soll M entweder unendlich weiterlaufen oder in einem vom Endzustand verschiedenen Zustand terminieren.

Alle anderen Präzisierungen des Begriffs **Berechenbarkeit** einer Funktion haben sich als äquivalent zu dem Begriff der TURING Berechenbarkeit erwiesen.[6] Die Äquivalenz der unterschiedlichen Ansätze zur Beschreibung der Berechenbarkeit bezieht sich auf die Klasse von Funktionen, die berechnet werden können. Aus diesem Grund formulierte ALONZO CHURCH im Jahre 1936 die **Church-sche These**,[7], die aussagt, dass die Klasse der TURING berechenbaren Funktionen mit der Klasse der intuitiv berechenbaren Funktionen übereinstimmt.

Beispiel [5.71]

In diesem Beispiel geben wir eine `TM` M an, die die Nachfolgefunktion für natürliche Zahlen berechnet. Die Nachfolgefunktion für natürliche Zahlen ordnet jeder natürlichen Zahl n den Nachfolger $n+1$ zu:

$$s(n) = n + 1.$$

Zur Darstellung der natürlichen Zahlen verwenden wir die unäre Codierung, dabei wird die Zahl $n \in \mathbb{N}$ durch $n+1$ Striche | dargestellt, also

$$\begin{array}{lcl} 0 & \longleftrightarrow & | \\ 1 & \longleftrightarrow & || \\ 2 & \longleftrightarrow & ||| \\ 3 & \longleftrightarrow & |||| \end{array}$$

usw. Das Inputalphabet ist also

$$\Sigma = \{\,|\,\}.$$

Die Nachfolgefunktion $s(n)$ wird durch die folgende TURING–Maschine berechnet, die an eine beliebige Folge von | noch ein weiteres Symbol | anhängt.

$$M = (\{q_0, q_1\}, \Sigma, \delta, q_0, q_1)$$

[6] Es gibt eine Reihe unterschiedlicher Ansätze, um die Berechenbarkeit formal beschreiben zu können. Dazu zählen

- λ–Kalkül von ALONZO CHURCH ([68], [69], Kap. 3.2.5, [91, 105])
- Registermaschinen
- Kellerautomaten mit zwei Stackspeichern ([54], Kap. 1.3)
- Automatenmodell von EMIL POST
- Theorie der μ–rekursiven Funktionen ([69], Kap. 3.2.2,)
- Termersetzungssysteme ([39], Chapter 13.3)
- POST–Systeme ([39], Chapter 13.2, [115, 116])
- Kombinatorische Logik ([96, 120]).
- Zelluläre Automaten (entwickelt in den 1940er Jahren von STAN ULAM und JOHN VON NEUMANN)

[7] Dies wird oftmals auch als **Church–Turing These** bezeichnet.

und der Übergangsfunktion

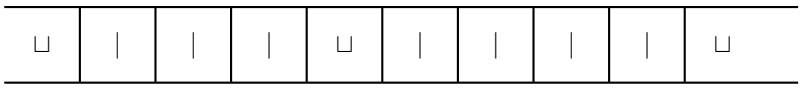
$$\begin{array}{lllll} (1) & q_0\ | & \longmapsto & q_0\ R \\ (2) & q_0\ \sqcup & \longmapsto & q_1\ | \end{array}$$

Betrachte den Fall $n = 3$. Die TM M arbeitet die folgenden Konfigurationen ab:

$$\begin{array}{rcl} q_0\ ||||\ \sqcup\sqcup & \xrightarrow{(1)} & |\ q_0\ |||\ \sqcup\sqcup \\ & \xrightarrow{(1)} & ||\ q_0\ ||\ \sqcup\sqcup \\ & \xrightarrow{(1)} & |||\ q_0\ |\ \sqcup\sqcup \\ & \xrightarrow{(1)} & ||||\ q_0\ \sqcup\ \sqcup \\ & \xrightarrow{(2)} & ||||\ q_1\ |\ \sqcup \end{array}$$

Beispiel [5.72]

In diesem Beispiel geben wir eine TURING Maschine an, die die Summe zweier natürlichen Zahlen berechnet. Wir verwenden wieder das unäre Alphabet, um die natürlichen Zahlen zu codieren. Die grundlegende Idee, wie die Maschine arbeitet, kann man sich anhand der folgenden Abbildung klar machen.



Die Anfangskonfiguration besteht darin, dass zwei Zahlen auf dem Eingabeband stehen, die durch das Blankzeichen ⊔ getrennt sind. In obiger Skizze sind das die beiden Zahlen 2 und 3. Wenn man einfach das Leerzeichen zwischen den beiden Zahlen löscht und durch | ersetzt, ergeben sich acht Striche, was die Zahl 9 codiert. Wenn dann zuletzt die beiden | ganz rechts gelöscht werden, ergibt sich die korrekte Summe.

Die TURING Maschine M, die dies umsetzt ist:

$$M = (Q, \Sigma, \delta, q^s, q^f)$$

mit

$$Q = \{q_0, q_1, q_2, q_3, q_4\}, \Sigma = \{|\}, q^s = q_0, \text{ und } q^f = q_4,$$

und den Übergängen

q_0 $\mid$	$\longmapsto$	q_0 R	nach rechts zum $\sqcup$ zwischen w_1 und w_2
q_0 $\sqcup$	$\longmapsto$	q_1 $\mid$	ersetze $\sqcup$ durch $\mid$
q_1 $\mid$	$\longmapsto$	q_1 R	nach rechts bis Ende w_2
q_1 $\sqcup$	$\longmapsto$	q_2 L	zurück zum letzten $\mid$
q_2 $\mid$	$\longmapsto$	q_2 $\sqcup$	letztes $\mid$ löschen
q_2 $\sqcup$	$\longmapsto$	q_3 L	zurück auf vorletztes $\mid$
q_3 $\mid$	$\longmapsto$	q_4 $\sqcup$	vorletztes $\mid$ löschen und terminieren

Wir betrachten als Beispiel die Addition von 3 und 4. Die Maschine durchläuft die folgenden Konfigurationen:

$$\begin{aligned}
q_0\ |||| \sqcup ||||| &\xrightarrow{4x(1)} |||| q_0\sqcup ||||| \\
&\xrightarrow{(2)} |||| q_1 |||||| \sqcup \\
&\xrightarrow{6x(3)} ||||||||| q_1\sqcup \\
&\xrightarrow{(4)} |||||||| q_2 | \sqcup \\
&\xrightarrow{(5)} |||||||| q_2 \sqcup \sqcup \\
&\xrightarrow{(6)} ||||||| q_3 | \sqcup\sqcup \\
&\xrightarrow{(7)} \underbrace{|||||||}_{8 \text{ mal}} q_4 \sqcup \sqcup\sqcup
\end{aligned}$$

Mit einer analogen Argumentation wie in Kapitel [1.2] kann man sehen, dass es Funktionen geben muss, die nicht TURING berechenbar sind. Die Menge der TURING berechenbaren Funktionen ist verschwindend gering im Vergleich zur Menge der existierenden Funktionen.

Es gibt nur abzählbar viele Programme einer Programmiersprache, die nur abzählbar viele Funktionen berechnen können. Andererseits gibt es bereits überabzählbar viele zahlentheoretische Entscheidungsfunktionen der Form

$$f : \mathbb{N} \longrightarrow \{0, 1\}.$$

Diese Funktionen sind durch den Werteverlauf charakterisiert, der eine unendliche 0–1–Folge darstellt. Die unendlichen 0–1–Folgen können als Folge der Nachkommastellen von reellen Zahlen des Intervalls $(0, 1)$ in Binärdarstellung interpretiert werden und davon gibt es überabzählbar viele.

Funktionen, die nicht TURING berechenbar sind, sind trotz ihrer großen Anzahl nicht einfach zu charakterisieren. Siehe dazu die Literatur.

$\Longrightarrow$ Übungen [5.49]

5.4 Gödelisierung von Turingmaschinen

Im Abschnitt [1.2.2] haben wir die Gödelisierung kennen gelernt, das ist ein systematisches Verfahren, einem Wort einer formalen Sprache eine Zahl (GÖDEL Zahl) zuzuordnen.

Eine wichtige Anwendung dieses Verfahrens besteht darin, dass es mit der Gödelisierung möglich ist, die Menge der TURING Maschinen abzuzählen.

Sei

$$M = (Q, \Sigma, \delta, q^s, q^f)$$

eine TURING Maschine. Die Zustandsmenge Q und das Bandalphabet Σ mit den Kopfbewegungen sind endliche Mengen, die wir durchnummerieren können, *i.e.*

$$Q = \{q_0, q_1, \ldots, q_k\}$$

und

$$\Sigma \cup \{L, R\} = \{a_1, a_2, \ldots, a_l, L, R\}.$$

Die Elemente dieser Mengen können wir binär darstellen, wir benutzen dazu einfach die Binärdarstellung der Indices der Mengenelemente, die Kopfbewegung L setzen wir $l+1$ und R setzen wir $l+2$. Beispielsweise wird der fünfte Zustand q_4 dargestellt durch die Binärzahl `101`.

Die Übergangsfunktion δ hat die folgende Form

$$\delta(q_i, a_j) = (q_k, a_l) \quad \text{mit } q_i, q_j \in Q; a_j, a_l \in \Sigma \tag{5.2a}$$

oder

$$\delta(q_i, a_j) = (q_k, B), \quad B = \{L, R\}, \tag{5.2b}$$

wobei B die Kopfbewegung darstellt. Wie haben die TURING Maschine so definiert, dass pro Schritt entweder ein Zeichen auf dem Band ersetzt wird, oder der Schreib–Lesekopf bewegt sich eine Zelle nach links oder nach rechts. Es gibt verschiedene Möglichkeiten, wie man eine TURING Maschine codiert.

Binärdarstellung

Um die Übergansgfunktion (5.2a) und (5.2b) im Binärsystem zu codieren, können wir folgendermaßen vorgehen. Wir codieren einen Übergang durch ein Wort über dem Alphabet $\{0, 1, \#\}$. Die Zustände und die Inputzeichen werden wie oben erwähnt, durch die Binärdarstellung ihrer Indices dargestellt. Die einzelnen Elemente trennt man durch das Symbol #. Damit können wir Anweisungen der Form (5.2a) oder (5.2b) wie folgt codieren:

$$\delta(q_i, a_j) = (q_k, a_l) \longrightarrow \#\#bin(i)\#bin(j)\#bin(k)\#bin(l).$$

Damit man sich auf das binäre Alphabet zur Codierung beschränken kann, betrachtet man eine Abbildung der Menge $\{0, 1, \#\}$ auf die Menge $\{\mathtt{0}, \mathtt{1}\}$:

$$0 \longmapsto \mathtt{00}, 1 \longmapsto \mathtt{01}, \# \longmapsto \mathtt{10}$$

Beispiel [5.73]

Wir haben im Beispiel [5.71] gezeigt, dass Nachfolgefunktion $s(n) = n+1$ durch die folgende TURING–Maschine berechnet wird:

$$M = (\{q_0, q_1\}, \Sigma, \delta, q_0, q_1)$$

und der Übergangsfunktion

$$\begin{array}{lcl} q_0 \mid & \longmapsto & q_0\ R \\ q_0 \sqcup & \longmapsto & q_1\ \mid, \end{array}$$

Die Menge der Zustände $Q = \{q_0, q_1\}$ können wir einfach durch

$$q_0 \longleftrightarrow \mathtt{0}, \quad q_1 \longleftrightarrow \mathtt{1}$$

codieren, das Bandalphabet $\Sigma \cup \{L, R\} = \{\mid, \sqcup, L, R\}$ durch

$$\begin{array}{rcl} \mid & \longleftrightarrow & 00 \\ \sqcup & \longleftrightarrow & 01 \\ L & \longleftrightarrow & 10 \\ R & \longleftrightarrow & 11 \end{array}$$

Dann können wir die beiden Anweisungen codieren in der Form

$$\#\#0\#00\#0\#11\#\#0\#01\#1\#00$$

Anhand des doppelten # Zeichen erkennt ma, dass eine neue Anweisung folgt.

In reiner Binärdarstellung ist dies

$$\mathtt{0101\,0001\,0000\,0100\,1001\,0110\,1000\,1000\,0111\,0110\,0000}$$

Interpretiert man dies als Binärzahl, so entspricht dies der Dezimalzahl

$$5\,567\,509\,202\,528.$$

Alternative Binärdarstellung

Um TURING Maschinen effektiv aufzählen zu können, müssen wir festlegen, wie TURING Maschinen codiert werden, wie sie letztendlich durch Worte über einem Alphabet dargestellt werden können. Es reicht aus, lediglich die Übergangsfunktion δ — also das Programm — zu codieren, wenn die anderen Bestandteile der Maschine wie Q, Σ, q^s, q^f aus der Codierung rekonstruiert werden können. Das geeignete Codealphabet ist das

binäre Alphabet $\Sigma = \{0, 1\}$, da das Bandalphabet Σ einer TM dieses als Bestandteil hat. Dann ist jede TURING Maschine in der Lage, Codierungen einer anderen Maschine zu lesen und zu verarbeiten.

Sei

$$M = (Q, \Sigma, \delta, q^s, q^f)$$

eine beliebige Standard TURING Maschine. Ist

$$\delta(q_i, a_j) = (q_k, a_l) \text{oder } \delta(q_i, a_j) = (q_k, D), \qquad D \in \{L, R\} \tag{5.3}$$

eine Anweisung des TURING Programms. Wir nehmen an, das Bandalphabet hat n Zeichen. Dann können wir die Bewegungen des Schreib–Lesekopfes als Zeichen $a_{n+1} = L$ und $a_{n+2} = R$ interpretieren, Die Instruktion (5.3) können wir durch das Wort

$$K = 0^i 1 0^j 1 0^k 1 0^j$$

codieren.

Wir können auf diese Weise jede Instruktion des TURING Programms als 01–Folge codieren und erhalten dadurch eine Reihe von Codeworten

$$K_1, K_2, K_3, \ldots$$

Aus diesen Codewörten kann der Code $< T >$ einer TURING Maschine M folgendermaßen festgelegt werden:

$$< T >= 111 K_1 11 K_2 11 \cdots 11 K_r 111. \tag{5.4}$$

Wir können (5.4) als Binärdarstellung einer natürlichen Zahl $n \in \mathbb{N}$ interpretieren, diese Zahl heisst **Index der Turing Maschine** T. Man beachte, dass es natürliche Zahlen gibt, die kein Index einer Maschine darstellen, da nicht die erforderliche Struktur gegeben ist. Um dies zu umgehen, benutzen wir die Konvention, dass jede Zahl, deren Binärdarstellung nicht die erforderliche Struktur (5.4) hat, der Index einer spezielle TURING Maschine ist, die wir leere TURING Maschine nennen. Das Programm dieser Maschine ist überall nicht definiert, *i.e.* für jedes Paar Zustand und Eingabesymbol gibt es keine Nachfolger. Für jeden Input terminiert die leere TURING Maschine in 0 Schritten.

Hieraus erhalten wir die Aussage:

> Jede natürliche zahl ist der Index einer TURING Maschine.

Primzahldarstellung

Eine andere Art der Codierung geht über die Primfaktoren, dies haben wir im Abschnitt [1.2.2] gesehen. Wir nummerieren die Zustände wieder durch

$$q_0 \longrightarrow 1, q_1 \longrightarrow 2, \ldots, q_k \longrightarrow k + 1$$

(damit vermeidet man die 0), analog setzen wir für das Bandalphabet und die Kopfbewegungen

$$a_1 \longrightarrow 1, a_2 \longrightarrow 2, \ldots, a_l \longrightarrow l, L \longrightarrow l+1, R \longrightarrow l+2.$$

Die Übergänge schreiben wir einfach als 4–Tupel: Die TURING Maschine, die den Nachfolger berechnet, hat die Übergänge

$$\begin{array}{lcll} q_0\ | & \longmapsto & q_0\, R & \Longleftrightarrow \quad q_0 \mid q_0 R \\ q_0 \sqcup & \longmapsto & q_1\ | & \Longleftrightarrow \quad q_0 \sqcup q_1 \mid \end{array}$$

Wir können die Zustände und die Zeichen nach dem oben angegebenen Schema wie folgt codieren:

$$\begin{aligned} q_0 &\longrightarrow 1 \\ q_1 &\longrightarrow 2 \\ \mid &\longrightarrow 1 \\ \sqcup &\longrightarrow 2 \\ L &\longrightarrow 3 \\ R &\longrightarrow 4 \end{aligned}$$

Dadurch wird das 4–Tupel $q_0 \mid q_0 R$ codiert durch

$$q_0 \mid q_0 R \quad \longrightarrow 1, 1, 1, 4$$

und das 4–Tupel $q_0 \sqcup q_1 \mid$ durch

$$q_0 \sqcup q_1 \mid \quad \longrightarrow 1, 2, 2, 1.$$

Die komplette Information über die TURING Maschine kann dann durch eine endliche Folge von Zahlen angegeben werden:

$$1, 1, 1, 4, 1, 2, 2, 1.$$

Noch kompakter kann diese Folge durch eine einzige Zahl ausgedrückt werden — das ist die GÖDEL Nummer dieser TURING Maschine:

$$2^1 \cdot 3^1 \cdot 5^1 \cdot 7^4 \cdot 11^1 \cdot 13^2 \cdot 17^2 \cdot 19^1 = 735\,265\,801\,070.$$

Nicht jede positive Zahl repräsentiert eine `TM`; Ob eine gegebene positive Zahl eine TURING Maschine repräsentiert oder nicht hängt davon ab, was die Folge der Exponenten in der Primfaktorzerlegung liefert, und nicht jede Zahlenfolge stellt eine `TM` dar. Diejenigen Folgen die eine Darstellung einer `TM` sind, haben als Länge ein Vielfaches der Form $4n$ von 4.

Wie auch immer, man erhält durch die obige Codierung eine lückenhafte Auflistung alle möglichen TURING Maschinen, in der jede (sinnvolle) TURING Maschine mindestens einmal auftaucht. Füllt man die Lücken, ergibt sich eine lückenlose Liste aller mögliche (und unmöglichen) TUING Maschinen

$$\mathtt{TM}_1, \mathtt{TM}_2, \mathtt{TM}_3, \mathtt{TM}_4, \ldots$$

und damit eine List alle einstelligen TURING berechenbaren Funktionen

$$f_1, f_2, f_3, \ldots$$

mit f_i wird von der Maschine $\texttt{TM}_i$ berechnet.

Betrachte beispielweise die TURING Maschine $\texttt{TM}_{210}$, gegeben durch die Folge $(1,1,1,1)$ oder

$$2^1 \cdot 3^1 \cdot 5^1 \cdot 7^1 = 210.$$

Die Maschine hat genau eine Anweisung in der Übergangsfunktion, sie lautet

$$(q_0, |, q_0\ |) \text{ oder } \delta(q_0, |) = (q_0, |)$$

5.4.1 Universelle Turing Maschine

In der obigen Definition [5.26] einer TURING Maschine ist das Programm — das ist die Übergangsfunktion — fest in die Maschine eingebaut und kann nicht verändert werden. Kodiert man die Beschreibung einer TURING–Maschine als hinreichend einfache Zeichenkette, so kann man eine sogenannte **universelle Turingmaschine** — dies ist selbst wieder eine TURING Maschine — konstruieren, welche eine Kodierung einer beliebigen TURING Maschine als Teil ihrer Eingabe nimmt und das Verhalten der kodierten TURING Maschine auf der ebenfalls gegebenen Eingabe simuliert. Aus der Existenz einer solchen universellen TURING Maschine folgt die Unentscheidbarkeit des Halteproblems. Eine ähnliche Idee, bei der das Programm als ein Teil der veränderbaren Eingabedaten betrachtet wird, liegt auch fast allen heutigen Rechnerarchitekturen zugrunde (VON–NEUMANN–Architektur).

Formal ist eine universelle Turingmaschine eine Maschine $\texttt{UTM}_k$, die eine Eingabe $w \,||\, x$ liest. Das Wort w ist hierbei eine hinreichend einfache Beschreibung einer TURING Maschine M_w, die zu einer bestimmten Funktion mit Eingabe x die Ausgabe berechnet. $||$ ist ein Trennzeichen zwischen Programmbeschreibung und Eingabe. Die universelle TURING Maschine $\texttt{UTM}_k$ simuliert also das Verhalten von M_w mit Hilfe der Funktionsbeschreibung w und der Eingabe x.

5.4.2 Das Halteproblem

5.5 Komplexitätsuntersuchungen

Die **Komplexitätstheorie** beschäftigt sich in erster Linie damit, die Leistungsfähigkeit von Computerprogrammen zu analysieren und Problemstellungen hinsichtlich des Aufwands zu klassifizieren, der zu ihrer Lösung durch ein Computeprogramm nötig ist.

Der Aufwand zur Lösung eines Problems mit einem Computer kann unter verschiedenen Gesichtspunkten charakterisiert werden, von denen der Speicherplatzbedarf und die Laufzeit die wichtigsten sind. Für die Betrachtungen hier beschränken wir uns auf die Analyse der Laufzeit.

Für diese Untersuchungen bildet das theoretische Modell der TURING Maschine die Grundlage. Gemäß der CHURCHschen These kann alles, was überhaupt intuitiv berechnet werden kann, auch mit einer TURING Maschine berechnet werden. Mit den **universellen Turing Maschinen** können die heutigen programmierbaren Rechner simuliert werden, so dass TURING Maschinen auch als einfaches Modell heutiger (und aller zukünftigen) Rechner verwendet werden können.

Die heutigen Rechner unterscheiden sich bezüglich der Geschwindigkeit nicht fundamental von den TURING Maschinen, denn alles was ein Computer leistet, kann eine TURING Maschine in einer Zeit leisten, die nur um einen polynomialen Faktor größer ist. Dies bedeutet genauer, dass wenn ein Rechner für eine Aufgabe — *e.g.* das Sortieren von n daten — eine Laufzeit benötigt, die durch eine Funktion $f(n)$ beschränkt ist,[8] dann benötigt eine TURING Maschine für die gleiche Aufgabe höchstens die Zeit $p(n) \cdot f(n)$, wobei

$$p(n) = \sum_{i=0}^{k} a_i n^i$$

ein Polynom vom Grad k ist mit nichtnegativen ganzzahligen Koeffizienten $a_i, i = 0, \ldots, k$..

5.5.1 Komplexitätsmaße

Die Laufzeit eines Programms bzw. einer TURING Maschine ist ein **Komplexitätsmaß**, das mit Hilfe einer Funktion $f(n)$ angegeben wird. Die Funktion f stellt den Wert des Komplexitätsmaßes in Abhängigkeit von der **Problemgröße** dar. Ein solches komplexitätsmaß ist nur sinnvoll für Programme, die verschiedene Eingaben in gleicher Weise verarbeiten können, beispielweise Sortier– oder Suchprogramme, oder Programme, die zahlentheoretische Funktionen berechnen.[9]

[8] Dies bedeutet, selbst im schlimmsten Fall (worst case) beträgt der Zeitaufwand bei einer Eingabe der Größe n höchstens $f(n)$ Zeiteinheiten.

[9] Ein Beispiel dafür ist der erweiterte EUKLIDische Algorithmus, der den `ggT` zweier Zahlen oder das multiplikative Inverse berechnet.

Bei TURING Maschinen definietr man alsn Problemgröße die Anzahl der Symbole, aus denen die Eingabe besteht. Zur Ermittlung der Laufzeit einer TURING maschine zählt man der Einfachheit halber nur die Anzahl der elementaren Anweisungen, die die Maschine ausführt.

Bei der Berechnung von Funktionen über Zahlen hängt die Laufzeit von der verwendeten Zahlendarstellung ab, denn die Zahlen, die Als Argumente eingegeben werden müssen in der Regel Ziffer für Ziffer von der Maschine gelesen werden, damit der Funktionswert berechnet werden kann. Verwendet man zur darstellung von natürlichen Zahlen die unäre Darstellung, die nach dem additiven Prinzip aufgebaut ist, so ist die Laufzeit bedeutend größer als bei Verwendung von Stellenwertsystemen, wie das Binär– oder Dezimalsystem.

Für die Zahl 109 müssen beispielweise bei der unären Darstellung 110 Striche eingelesen werden, in dezimaler Schreibweise sind das nur drei Ziffern, binär auch nur sieben. Allgemein müssen für eine natürliche Zahl $n, n \geq 0$ in der unären codierung $n + 1$ Ziffern gelesen werden. Bei der dezimalen Darstellung sind dies

$$\lfloor \log_{10} n \rfloor + 1$$

Ziffern, falls $n > 0$ ist.[10]

Die Binärdarstellung von 109 ist:

$$109_{10} = 1\,101\,101_2,$$

generell hat eine Binärzahl

$$\lfloor \log_2 n \rfloor + 1$$

Binärziffern. Der Unterschied zwischen der binären und dezimalen Darstellung ist lediglich ein konstanter Faktor, denn Logarithmen lassen sich ineinander umrechnen gemäß

$$\log_{10} n = \log_{10} 2 \cdot \log_2 n,$$

wobei $\log_{10} 2$ ein konstanter Faktor ist. Um den praktischen, relevanten Problemen bei der theoretischen Analyse möglichst nahe zu kommen, werden in der Komplexitätstheorie Binärdarstellungen von Zahlen zugrunde gelegt.

Der Wert eines Komplexitätsmaßes hängt nicht nur von der Problemgröße ab sindern auch von den Eingabedaten selbst.

Sucht man beispielweise das erste Vorkommen von Wörtern in einem Text, so wird man als Problemgröße die Länge des Textes und die Länge des zu suchenden Wortes definieren. Bei ein und demselben Eingabetextes spielt aber auch die Häufigkeit des Auftretens des gesuchten Wortes eine Rolle. So wird man im Normalfall in einem deutschsprachigen Text viel schneller ein Vorkommen des Wortes *die* finden als ein Vorkommen des Wortes *Zoo*.

[10] Die GAUSS–Klammer $\lfloor x \rfloor$ ist die größte ganze Zahl kleiner oder gleich x.

Daher müssen komplexitätsmaße relativiert werden, dies erfolgt dadurch, dass man sie nach gewissen Klassen von Eingaben differenziert, Es werden drei Fälle unterschieden:

1. **Ungünstigster Fall, worst case** Welches ist der Maximalwert, den $f(n)$ bei Eingaben der Größen n annehmen kann?
2. **Günstigster Fall, best case** Welches ist der Minimalwert, den $f(n)$ bei Eingaben der Größen n annehmen kann?
3. **Normalfall, average case** Welches ist der Erwartungswert von $f(n)$ für Eingaben der Größen n?

in der Komplexitätstheorie spielen Komplexitätsmaße in Form eines Polynoms eine dominante Rolle. Daher die folgende Definition.

Definition [5.29]:

Ein **Polynom** ist eine Funktion

$$p : \mathbb{N} \longrightarrow \mathbb{N}$$

mit

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n^1 + a_0.$$

Hier ist $k \in \mathbb{N}$ und die Koeffizienten des Polynoms a_i sind alle natürliche Zahlen. Das größte i mit $a_i \neq 0$ heißt der Grad des Polynoms.

5.5.2 Die Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$

Turing Maschinen werden weniger zur Untersuchung der Laufzeit spezieller Algorithmen verwendet, sondern hauptsächlich zur Unterscheidung von **Problemklassen** bezÜglich des Zeitaufwands, der zur Lösung der Probleme erforderlich ist.

Der Begriff **Problem** betrachten wir als Zusammenfassung gleichartiger Problemausprägungen, zu denen es jeweils eine spezifische Lösung gibt. Man denke dabei an Sortierprobleme, bei denen eine Problemausprägung aus einer konkreten Datenmenge besteht, die sortiert werden soll. Die spezifische LÖsung ist dann die sortierte Datenmenge.

Wir betrahten nun ausschließlich Problemen die durch Algorithmen gelöst werden. Algorithmen kann man abstrakt als Verfahren zur Berechnung einer partielle Funktion

$$f : E \longrightarrow A$$

von der Menge der zulässigen Eingaben[11] E in dei Menge der Ausgaben — oder Lösungen — auffassen. Wegen der Partialität muss nicht zu jeder Problemausprägung eine Lösung existieren. In diesem Fall stoppt der Algorithmus in einem vom Endzustand verschiedenen Zustand oder läuft unendlich weiter. Man erhält dann im Prinzip einen Erkennungsalgorithmus für die Sprache, die aus allen Eingaben besteht, für die f definiert ist, denn genau für diese Eingaben terminiert der Algorithmus nach endlich vielen Schritten im Endzustand.

Bei Komplexitätsuntersuchungen von Algorithmen zur Lösung von Problemen kann man sich daher auf das Erkennen von Sprachen durch TURING maschinen beschränken. Wir benutzen dazu nichtdeterministsiche TURING Maschinen. Diese sind — wie bei den endlichen Automaten — gleich mächtig wie die deterministischen `TM`.

Definition [5.30]:

Eine **nichtdeterministische Turing–Maschine** `TM` ist ein 5–Tupel

$$M = (Q, \Sigma, \delta, q^s, q^f)$$

mit:

Q ist eine endliche, nichtleere Menge, die Menge der Zustände

Σ ist eine endliche, nichtleere Menge, das Maschinenalphabet

δ ist die Übergangsirelation

$$\delta \subseteq \left(Q \setminus \{q^f\}\right) \times (\Sigma \cup \{\sqcup\}) \times Q \times (\Sigma \cup \{\sqcup\} \cup \{L, R\})\,. \quad (5.5)$$

Das Zeichen $\sqcup$ steht für Blank (=Leerzeichen). Beachte gemäß dieser Definition kann die Maschine das Blank lesen und schreiben. L steht für 'Links' und R steht für 'Rechts'.

q^s ist der Startzustand, $q^s \in Q$.

q^f ist der Accept– oder Endzustand, $q^f \in Q$.

Die von einer nichtdeterministischen TURING Maschine erkannte bzw. akzeptierte Sprache ist die Menge aller Worte, für die es mindestens einen akzeptierenden Verlauf gibt.

Nichtdeterministische TURING Maschinen arbeiten völlig analog zu deterministischen TURING Maschinen. Der Unterschied besteht darin, dass es für einen Arbeitsschritt mehrere Möglickeiten geben kann. Die äußert sich in Gl. (5.5)

[11] Das sind die möglichen Problemausprägungen.

dadurch, dass δ eine Teilmenge ist. Die Menge aller möglichen Verläufe bei der Abarbeitung eines Wortes durch eine nichtdeterministsiche TURING Maschine kann man sich als Baum vortsellen. Die Wurzel entspricht der Anfangskonfiguration, jeder Verarbeitungsschritt führt auf einen Pfad in die Tiefe, wenn es für den folgenden Schritt mehrere Möglichkeit gibt, verzweigt sich der Pfad.

Die Äquivalenz deterministischer und nichtdeterministische TURING Maschinen zeigt man dadurch, dass man die baumartig sich verzweigenden Berechnungsmöglichkeiten einer nichtdeterministischen TURING Maschine systematisch nach der Strategie *breadth first* durchspielt.[12]

Bei einer nichtdeterministischen TURING Maschine wir ddie Zeit, die zum Erkennen eines Wortes benötigt wird, als Laufzeit eines minimalen akzeptierenden Verlaufs definiert. Unter dieser voraussetzung gilt die folgende Definition für das Erkennen einer Sprache in polynomialer Zeit für beide Varianten von `TM`.

Definition [5.31]:

Eine TURING Maschine erkennt eine Sprache L in polynomialer Zeit, wenn es ein Polynom p gibt, so dass jedes Wort $w \in L$ der Länge n in einer Zeit t mit $t \leq p(n)$ erkannt wird.

Anmerkungen:

1. Der Begriff *Zeit* in der Definition [5.31] wird synonym mit *Anzahl von Elementarschritten* verwendet.

2. Ob ein Wort von einer nichtdeterministischen TURING Maschine in polynomialer Zeit erkannt wird, hängt davon ab, ob es mindestens einen erfolgreichen Pfad gibt, dessen Anzahl von Elementarschritten polynomial beschränkt sit.

Wir können nun zwei Klassen von Sprachen definieren, die von fundamentaler Bedeutung für die Komplexitätstheorie sind.

[12] Einen Beweis dieser Äquivalenz findet man in dem Buch von MICHAEL SIPSER [44], Theorem 3.10, oder HOPCROFT und ULLMAN [34], Satz 7.3.

Definition [5.32]:

Die Menge $\mathcal{P}$ besteht aus allen Sprachen, die von einer deterministischen TURING Maschine in polyomialer Zeit erkannt werden.

Die Menge $\mathcal{NP}$ besteht aus allen Sprachen, die von einer nichtdeterministischen TURING Maschine in polyomialer Zeit erkannt werden.

Anmerkung:

Eine häufig verwendete alternative Charakterisierung der Komplexitätsklasse $\mathcal{NP}$ lautet: $\mathcal{NP}$ ist die Klasse aller Entscheidungsprobleme, deren (bekanntes) Ergebnis in polynomialer Zeit *verifiziert* werden kann.

Die Probleme, die den Sprachen der Komplexitätsklasse $\mathcal{P}$ entsprechen, zeichnen sich dadurch aus, dass zu jeder Ausprägung des Problems ein schneller *i.e.* polynomialer, Lösungsweg existiert. Obwohl ein Polynom mit einem hohen Grad eine enorm große und praktisch nicht tolerierbeare Laufzeit bedeuten kann, werden solche Probleme in der Informatik als *effizient lösbar* bezeichnet.

Die zur Sprachklasse $\mathcal{NP}$ gehörenden Probleme, bei denen eine schnelle Lösung erst aufwändig gesucht werden muss, *i.e.* in einer Zeit, die nicht durch ein Polynom beschränkt werden kann, werden als nicht effizient auf einem Rechner lösbar betrachtet. Ein solcher Zeitaufwand wird beispielsweise durch die Funktion $f(n) = 2^n$ beschrieben, die für größer werdende n schneller wächst als jedes Polynom $p(n)$. Das durch $f(n) = 2^n$ beschriebene Wachstumsverhalten wird als **exponentielles Wachstum** bezeichnet.

5.5.3 Die O-Notation

Das grundlegende Konzept, um Funktionen miteinander vergleichen zu können, ist die sogenannte O–Notation. Diese Notation wurde von EDMUND LANDAU (1877 — 1938) in die Mathematik eingeführt. Die O–Notation ist folgendermaßen definiert:

Definition [5.33]:

Seien $f, g : \mathbb{N}_0 \longrightarrow \mathbb{R}_+$ zwei Funktionen. Die Funktion f ist von der Ordnung g, falls es eine Konstante $c > 0$ und eine natürliche Zahl $n_0 \in \mathbb{N}$ gibt mit

$$f(n) \leq c \cdot g(n) \; \forall \, n \geq n_0$$

Mit der Schreibweise $O(g)$ bezeichnet man die Menge der Funktionen, die von der Ordnung g sind:

$$O(g) = \{f : \mathbb{N}_0 \to \mathbb{R}_+ \mid \exists\, c > 0 \text{ und } n_0 \in \mathbb{N}_0 \text{ mit } f(n) \leq c \cdot g(n) \,\forall\, n \geq n_0\}$$

Falls $f \in O(g)$ ist, sind also alle Funktionswerte von f ab einer festen Stelle n_0 kleiner als die entsprechenden Funktionswerte der Funktion g (abgesehen von der Konstanten). Ab diesem n_0 wächst f höchstens so schnell wie g.

Anstatt der mathematisch exakten Notation $f \in O(g)$ findet man häufig die etwas laxere Schreibweise $f = O(g)$. Dies ist synonym zu lesen mit f *ist von der Ordnung* g. Denn das Gleichheitszeichen drückt nicht die Gleichheit aus, denn f ist eine Funktion und $O(g)$ ist eine Menge von Funktionen.

Beispiel [5.74]

Wir zeigen in diesem Beispiel, dass die Funktion

$$f(n) = 3n^2 + 7n + 11$$

von der Ordnung n^2 ist, *i.e.* es gilt:

$$3n^2 + 7n + 11 \in O(n^2)$$

Gemäß unserer Definition muss für $f(n) = 3n^2 + 7n + 11$ und $g(n) = n^2$ gezeigt werden, dass es ein $c > 0$ und ein n_0 gibt, so dass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$.

Man kann das Polynom zweiten Grades $f(n)$ folgendermaßen abschätzen:

$$\begin{aligned} f(n) &= 3n^2 + 7n + 11 \\ &\leq 3n^2 + 7n^2 + 11n^2 \quad \forall\, n \geq 0 \\ &\leq 11n^2 + 11n^2 + 11n^2 \quad \forall\, n \geq 0 \\ &\leq 33n^2 \quad \forall\, n \geq 0 \\ &= 33 \cdot g(n) \end{aligned}$$

Damit haben wir also ein $c = 33 > 0$ und ein $n_0 = 0$ gefunden, so dass

$$f(n) \leq c \cdot g(n) \quad \text{für alle } n \geq n_0$$

gilt. Damit ist gezeigt, daß

$$3n^2 + 7n + 11 \in O(n^2)$$

mit anderen Worten, das Polynom zweiten Grades $3n^2 + 7n + 11$ wächst im Wesentlichen wie n^2.

Theorem [5.19]:

Für $k \in \mathbb{N}_0$ sei $p_k : \mathbb{N}_0 \to \mathbb{R}_+$ definiert durch

$$p_k = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \cdots + a_1 \cdot n + a_0$$
$$\sum_{l=0}^{k} a_l \cdot n^l, \quad a_i \in \mathbb{R}, \quad 0 \leq i \leq k,\ k > 0$$

ein Polynom vom Grad k über den reellen Zahlen. Dann gilt:

$$p_k \in O(n^k).$$

Beweis:

Das Verfahren, diese Aussage zu beweisen, beruht auf einer ähnliche Abschätzung wie im obigen Beispiel:

$$\begin{aligned} p_k &= a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \cdots + a_1 \cdot n + a_0 \\ &\leq a \cdot n^k + a \cdot n^{k-1} + \cdots + a \cdot n + a \quad \text{mit } a = \max\{|\, a_i \,|\, 0 \leq i \leq k\} \\ &\leq \underbrace{a \cdot n^k + a \cdot n^k + \cdots + a \cdot n^k}_{k+1 \text{ mal}} \quad \text{für alle } n \geq 0 \\ &= a(k+1)n^k \qquad \text{für alle } n \geq 0 \end{aligned}$$

Setzt man nun

$$c = a \cdot (k+1)$$

dann gilt:

$$p_k \leq c \cdot n^k \qquad \text{für alle } n \geq 0$$

und damit ist $p_k \in O(n^k)$.

Jedes Polynom vom Grad k wächst also wie n^k.

Rechenregeln

Seien $f, g, h : \mathbb{N}_0 \to \mathbb{R}_+$. Dann gilt:

1. $f \in O(f)$
2. $d \cdot f \in O(f)$ für $d \in \mathbb{R}_+$
3. $f + g \in O(g)$ falls $f \in O(g)$
4. $f + g \in O(\max\{f, g\})$
5. $f \cdot g \in O(f \cdot h)$ falls $g \in O(h)$

Beweis:

Zu 1.: Aus $f(n) \leq 1 \cdot f(n)$ für alle $n \geq 0$ folgt sofort die Behauptung.

Zu 2.: Aus der ersten Regel folgt $f \in O(f)$. Dies impliziert, daß es eine Konstante $c > 0$ und ein $n_0 \in \mathbb{N}$ gibt mit

$$f(n) \leq c \cdot f(n)$$

für alle $n \geq n_0$. Aus dieser Ungleichung folgt durch Multiplikation mit einer Konstanten $d \in \mathbb{R}_+$:

$$d \cdot f(n) \leq d \cdot c \cdot f(n)$$

für alle $n \geq n_0$. Daher gibt es eine Konstante $c' = d \cdot c$, so daß gilt:

$$d \cdot f(n) \leq c' \cdot f(n)$$

für alle $n \geq n_0$. Daher folgt: $d \cdot f \in O(f)$.

Zu 3.: Gemäß Voraussetzung ist $f \in O(g)$. Daher existiert ein $c > 0$ und ein $n \in \mathbb{N}_0$ mit

$$f(n) \leq c \cdot g(n)$$

für alle $n \geq n_0$. Daraus folgt:

$$\begin{aligned} f(n) + g(n) &\leq c \cdot g(n) + g(n) \quad \text{für alle } n \geq n_0 \\ &= (c+1) \cdot g(n) \quad \text{für alle } n \geq n_0 \\ &= c' \cdot g(n) \quad \text{für alle } n \geq n_0 \end{aligned}$$

Daraus folgt $f + g \in O(g)$.

Zu 4.: Die max-Funktion ist definiert über

$$\max\{f,g\}(n) = \begin{cases} f(n), & \text{falls } f(n) \geq g(n) \\ g(n), & \text{sonst} \end{cases}$$

Hieraus folgt:

$$\begin{aligned} f(n) + g(n) &\leq \max\{f,g\}(n) + \max\{f,g\}(n) \quad \text{für alle } n \geq n_0 \\ &= 2 \cdot \max\{f,g\}(n) \quad \text{für alle } n \geq n_0 \end{aligned}$$

Damit folgt $f + g \in O(\max\{f,g\})$.

Zu 5.: Da laut Voraussetzung $g \in O(h)$, folgt, es gibt ein $c > 0$ und ein $n_0 \in \mathbb{N}_0$ mit:

$$g(n) \leq c \cdot h(n)$$

für alle $n \geq n_0$.

Daraus folgt aber:

$$\begin{aligned} f(n) \cdot g(n) &\leq f(n) \cdot c \cdot h(n) \quad \text{für alle } n \geq n_0 \\ &= c \cdot f(n) \cdot h(n) \quad \text{für alle } n \geq n_0 \end{aligned}$$

Damit folgt $f \cdot g \in O(f \cdot h)$.

Die O–Notation ermöglicht also asymptotische Aussagen über *obere Schranken*.

5.5.4 Laufzeitverhalten

Die im Rahmen von Laufzeitbetrachtungen von Algorithmen wichtigsten Ordnungsfunktionen sind in der Tabelle [5.1] aufgelistet.

Wachstum	**Ordnung**	**Beispiel**
konstant	$O(1)$	Test: Zahl gerade/ungerade
logarithmisch	$O(\log n)$	bin-Search
linear	$O(n)$	lineare Suche
$n - log - n$	$O(n \log n)$	Quicksort
polynomiell	$O(n^k),\ k \geq 2$	Bubble Sort
exponentiell	$O(d^n), d > 1$	Primfaktorzerlegung

Tabelle 5.1: Ordnungsfunktionen von Algorithmen.

Um ein Gefühl dafür zu bekommen, welcher Rechenaufwand eines konkreten Rechners bei der Abarbeitung von Algorithmen aus den unterschiedlichen Ordnungsklassen anfällt, betrachten wir folgendes Szenario. Nehmen wir an, ein konkreter Rechner kann einen elementaren Schritt eines Algorithmus (Zuweisung, logischer Vergleich, arithmetische Operation) in einer Millisekunde abarbeiten (10^{-3} sec). Auf diesem Rechner werden nun Algorithmen $A_1, \ldots A_5$ aus den Ordnungsklassen $O(n), O(n \log n), O(n^2), O(n^3)$ und $O(2^n)$ abgearbeitet. Um einen Vergleich ziehen zu können, betrachten wir die Länge von Eingaben, die der Rechner in einer Sekunde, einer Minute und einer Stunde verarbeiten kann. Dies ist in Tabelle [5.2] aufgelistet.

Der Schritt von der polynomialen Laufzeit wie n^2 oder n^{10} zur exponentiellen Laufzeit ist enorm. Dies wird sehr deutlich, wenn man Werte unterschiedlicher Funktionen gegenüberstellt. Wir nehmen dazu vereinfachend einmal an, jeder Schritt kann in einer Mikrosekunde ($t_{Schritt} = 10^{-6}$ sec $= 1\mu s$) abgearbeitet werden.

In der Tabelle [5.3] haben die benutzten Einheiten folgende Bedeutung:

Alg	**Laufzeit**	**1 sec**	**1 min**	**1 h**
A_1	n	1.000	$6 \cdot 10^4$	$3.6 \cdot 10^6$
A_2	$n \log n$	128	4615	$2 \cdot 10^5$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Tabelle 5.2: Größe des Inputs, die pro Zeiteinheit verarbeitet werden kann bei unterschiedlichen Laufzeitverhalten von Algorithmen.

$\boldsymbol{n}$	$\log_2 n$	$\boldsymbol{n}$	$\boldsymbol{n} \log_2 n$	$\boldsymbol{n^2}$	$\boldsymbol{n^{10}}$	$\boldsymbol{2^n}$	$\boldsymbol{n!}$
1	0 μs	1 μs	0 μs	1 μs	1 μs	2 μs	1 μ s
10	3 μs	10 μs	33 μs	100 μs	3h	1 ms	4 s
20	4 μs	20 μs	86 μs	400 μs	119 d	1 s	77094 a
30	5 μs	30 μs	147 μs	900 μs	19 a	18 min	8 10^9 Mrd a
40	5 μs	40 μs	213 μs	1.6 ms	332 a	13 d	2 10^{25} Mrd a
50	6 μs	50 μs	282 μs	2.5 ms	3095 a	36 a	10 10^{41} Mrd a
60	6 μs	60 μs	354 μs	3.6 ms	19161 a	36534 a	3 10^{59} Mrd a
70	6 μs	70 μs	429 μs	4.9 ms	89511 a	37 Mio a	4 10^{77} Mrd a
80	6 μs	80 μs	506 μs	6.4 ms	340248 a	38 Mrd a	2 10^{96} Mrd a
90	6 μs	90 μs	584 μs	8.1 ms	1 Mio a	39228 Mrd a	5 10^{115} Mrd a
100	7 μs	100 μs	664 μs	10 ms	3 Mio a	40 10^6 Mrd a	5 10^{135} Mrd a

Tabelle 5.3: Laufzeitverhalten verschiedener Problemklassen.

1 μs sind 10^{-6} Sekunden

1 ms ist eine Millisekunde sind 10^{-3} Sekunden

1 s ist eine Sekunde

1 min ist eine Minute

1 h ist eine Stunde

1 d ist ein Tag

1 a ist ein Jahr

Wenn die Laufzeit für eine Problemdimension n wie eine feste Potenz — etwa n^2 oder $10n^{15} + 13n^6$ wächst, dann läuft der Algorithmus in **polynomialer Zeit**. Man spricht von einem **Polynomialzeit Algorithmus**. Wächst die Laufzeit wie 2^n oder schneller — etwa $3^n + n^{100}$ oder gar $n!$ — dann läuft der Algorithmus in **exponentieller Zeit**.

Bei der Effizienz von Algorithmen gilt folgende Einteilung:

Definition:

Sei A ein Algorithmus dessen Input die Bitlänge l hat.

① A ist ein **Polynomialzeit Algorithmus**, wenn seine Laufzeit $O(l^c)$ ist für eine Konstante $c > 0$.

② A ist ein **Exponentialzeit Algorithmus**, wenn seine Laufzeit nicht von der Form $O(l^c)$ ist für jedes $c > 0$.

③ A ist ein **Subexponentialzeit Algorithmus**, wenn die Laufzeit $O(2^{o(l)})$ ist, und A ist kein Polynomialzeit Algorithmus.

④ A ist ein **Vollexponentialzeit Algorithmus**, wenn seine Laufzeit nicht die Form $O(2^{o(l)})$ hat.

Ein Subexponentialzeit Algorithmus ist ebenfalls ein Algorithmus der Kategorie Exponentialzeit Algorithmus, insbesondere sind Subexponentialzeit Algorithmen keine Polynomialzeit Algorithmen. Das Laufzeitverhalten der Subexponentialzeit Algorithmen liegt irgendwo zwischen Polynomialzeit und Exponentialszeit Algorithmen.

Sei A ein Algorithmus mit einer ganzen Zahl n als Input — oder eine kleine Menge von Zahlen modulo n. Dann ist die Inputgröße des Algorithmus

$$l = \lfloor \lg_2 n \rfloor + 1$$

Bits. Falls die Laufzeit des Algorithmus A die Form

$$L[\alpha, c] = O\Big(e^{(c+o(1))(\log n)^{\alpha}(\log\,\log n)^{1-\alpha}} Bigr)$$

hat, $c, \alpha = const$ mit $0 < \alpha < 1$, dann ist A subexponentiell.

5.5.5 Das $\mathcal{P} = \mathcal{NP}$–Problem

Ein bekanntes Problem der Klasse $\mathcal{NP}$ ist das **Erfüllbarkeitsproblem**, kurz **SAT–Problem**.[13] Das Erfüllbarkeitsproblem besagt, dass für jeden BOOLEschen Ausdruck entschieden werden soll, ob es eine erfüllende Wahrheitswertebelegung gibt oder nicht.

Beispielsweise ist der BOOLEsche Ausdruck

$$(\neq a \vee b) \wedge (a \vee \neg c \vee \neg b) \wedge \neg c \tag{5.6}$$

erfüllbar, *i.e.* für $a = c =$ falsch und $b =$ wahr nimmt der Ausdruck den Wert wahr an. Dagagen ist der BOOLEsche Ausdruck

$$a \wedge (\neg aeeb) \wedge (\neg b \vee \neg c) \wedge c$$

[13] Diese Bezeichnung kommt aus dem Englischen, SAT steht für satisfiable.

a	b	c	$(\neq a \vee b)$	$a \vee \neg b \vee \neg c)$	$\neq c\ A \wedge B \wedge C$	
f	f	f	w	w	w	w
f	f	w	w	w	f	f
f	w	f	w	w	w	w
f	w	w	w	f	f	f
w	f	f	f	w	w	w
w	f	w	f	w	f	f
w	w	f	w	w	w	w
w	w	w	w	w	f	f

Tabelle 5.4: Wahrheitswerte Tabelle eines BOOLEschen Ausdrucks.

nicht erfüllbar. Für jede beliebige Belegung der Variablen a, b, c mit den Werten `w` oder `f` ist der gesamte Ausdruck immer falsch `f`.

Ein offensichtlicher Algorithmus für das SAT–Problem setzt für einen beliebig vorgegebenen BOOLEschen Ausdruck systematisch alle möglichen Kombinationen von Wahrheitswerten ein. Dies erreicht man stets mit den Wahrheitswertetafeln in der Aussagenlogik, wir sehen in der nachfolgenden Tabelle [5.4], dass es Belegungen der Variablen a, b, c gibt, so dass der BOOLEsche Ausdruck (5.6) wahr wird.

Diesen Algorithmus kann man durch eine *nichtdeterministische* TURING Maschine realisieren, die der Reihe nach die Variablen mit Wahrheitswerten belegt. Für jede Variable gibt es zwei Möglichkeiten, daher ist die Abarbeitung darstellbar als binärer Baum. Wenn ein eingegebener BOOLEscher Ausdruck erfüllbar ist, entspricht mindestens ein Pfad des Baums dem erfolgreichen Abarbeiten des Ausdrucks. Man kann zeigen, dass eine erfolgreiche Verarbeitung einen Aufwand benötigt, der durch ein Polynom $p(n)$ beschränkt ist, wobei n die Länge des als Eingabewort codierten BOOLEschen Ausdrucks ist.[14] Dies impliziert, dass

$$SAT \in \mathcal{NP}.$$

Werden von einer *deterministischen* TURING Maschine systematisch sämtliche Möglichkeiten in der Tabelle [5.4] ausprobiert, so wird es immer worst case Fälle geben. Bei diesen Fällen steht erst nach dem Testen der letzten Kombination das Ergebnis fest. In diesen Fällen benötigt die TURING Maschine 2^k viele Schritte, um sämtliche 2^k mögliche Belegungen auszutesten, wobei k die Anzahl der verschiedenen Variablen des BOOLEschen Ausdrucks ist. Da die Funktion 2^k für wachsende k exponentiell anwächst, muss man einen grundlegend schnelleren Algorithmus finden, um nachweisen zu können, dass das SAT Problem in der Klasse $\mathcal{P}$ liegt.

[14] Eine technische Umsetzung dieser Aussagen findet man in dem Buch von WEGENER [47], Kapitel 3.3.

Ein schnellerer Algorithmus als das Ausprobieren aller möglichen Wertebelegungen wurde bisher nicht gefunden, das SAT Problem ist eines der meist untersuchten Probleme. Daher besteht die berechtigte Annahme, dass es tatsächlich keinen Polynomialzeit–Algorithmus für das SAT Problem gibt.[15] Würde man für das SAT Problem einen effizienten Algorithmus finden, dann kann man daraus für alle anderen Probleme aus der Klasse $\mathcal{NP}$ einen effizienten Algorithmus konstruieren. In diesem Fall stimmen die beiden Komplexitätsklassen überein, *i.e.*

$$\mathcal{P} = \mathcal{NP}.$$

Daher spielt das SAT Problem — wie auch andere Probleme — eine herausragende Rolle, die man mit der Eigenschaft **$\mathcal{NP}$–vollständig** cCharakterisiert.

Die Frage, ob $\mathcal{P} = \mathcal{NP}$ gilt, ist eines der berühmtesten noch offenen Probleme der Informatik/Mathematik [59]. Die überwiegende Mehrheit der Fachleute tendiert dazu, dass die beiden Klassen verschieden sind. Es gibt dann Probleme, zum Beispiel das SAT Problem, die prinzipiell auf einem Computer nicht effizient lösbar sind.

Beispiel [5.75]

Wir geben hier einige bekannte $\mathcal{NP}$ Probleme an.

- Das Problem des Handlungsreisenden (Traveling Salesman Problem, TSP) ist $\mathcal{NP}$ vollständig.
- Das Cliquenproblem ist $\mathcal{NP}$ vollständig.
- Das Rucksackproblem ist $\mathcal{NP}$ vollständig.

---

[15] Man beachte, es gibt aber bis dato *keinen* mathematisch rigorosen Beweis, dass es diesen *nicht* gibt.

5.6 Übungen

Übung 5.49:

Gegeben ist die folgende TURING Maschine

$$M = (Q, \Sigma, \delta, q^s, q^f)$$

mit

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\} \\ \Sigma &= \{\ |\ \}, \\ q^s &= q_0, \\ q^f &= q_8 \end{aligned}$$

und den Übergängen

$$\begin{array}{lcl} q_0\ | & \longmapsto & q_1\ \sqcup \\ q_1\ \sqcup & \longmapsto & q_2\ R \\ q_2\ | & \longmapsto & q_3\ \sqcup \\ q_3\ \sqcup & \longmapsto & q_4\ R \\ q_4\ | & \longmapsto & q_5\ \sqcup \\ q_5\ \sqcup & \longmapsto & q_0\ R \\ q_0\ \sqcup & \longmapsto & q_6\ | \\ q_6\ | & \longmapsto & q_6\ R \\ q_6\ \sqcup & \longmapsto & q_7\ | \\ q_7\ | & \longmapsto & q_7\ R \\ q_7\ \sqcup & \longmapsto & q_8\ | \\ q_2\ \sqcup & \longmapsto & q_8\ | \\ q_4\ \sqcup & \longmapsto & q_7\ | \end{array}$$

Diese TURING Maschine berechnet eine Funktion $f(n)$, wobei n eine natürliche Zahl in unärer Darstellung ist. Welche Funktion berechnet diese Maschine?

Übung 5.50:

Man gebe eine TURING Maschine an, die die Funktion

$$f(n) = 2n$$

berechnet, wobei n eine natürliche Zahl in unärer Darstellung ist.

Übung 5.51:

Geben Sie eine TURING Maschine an, die für natürliche Zahlen in Binärdarstellung die Nachfolgerfunktion $f(n) = n + 1$ berechnet.

Hinweis: Man überlegt sich anhand einfacher Beispiele, wie man die Nachfolger erhält. Dazu ist es zweckmäßig, Beispiele von geraden und ungeraden zahlen zu betrachten, *e.g.* `100110` oder `11001`.

Übung 5.52:

Es gibt TURING Maschinen, die, angesetzt auf das leere Eingabeband, eine endliche Folge von Strichen auf das band schreiben und anschließend im Endzustand terminieren. Solche TURING Maschinen heißen **Busy–Beaver Turing Maschinen**, denn man kann sich die Striche als Holzstämme vorstellen, die ein Biber für seinen Dammbau heranschleppt.

Gegeben ist die folgende TURING Maschine

$$M = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}, \{\,|\,\}, \delta, q_0, q_7)$$

mit den Übergängen δ:

$$\begin{array}{llll} q_0\,\sqcup & \longmapsto & q_3\,| & \\ q_0\,| & \longmapsto & q_2\,L & \\ q_1\,\sqcup & \longmapsto & q_4\,| & \\ q_1\,| & \longmapsto & q_7\,| & \\ q_2\,\sqcup & \longmapsto & q_5\,| & \end{array} \qquad \begin{array}{llll} q_2\,| & \longmapsto & q_6\,\sqcup & \\ q_3\,| & \longmapsto & q_1\,R & \\ q_4\,| & \longmapsto & q_2\,R & \\ q_5\,| & \longmapsto & q_0\,L & \\ q_6\,\sqcup & \longmapsto & q_1\,L & \end{array}$$

Beschreiben Sie die Arbeitsweise dieser Maschine, wenn sie auf das leere Band angesetzt ist. Wieviele Stämme schleppt der fleißige Biber zu seinem Damm?

Übung 5.53:

Geben Sie eine `TM` an, die die als Binärzahl interpretierte Bandinschrift $w \in \{\mathtt{0}, \mathtt{1}\}^*$ in die Zweierkomplementdarstellung umwandelt.

Übung 5.54:

Konstruieren Sie eine TURING Maschine M, die rechts neben der Eingabe $w \in \{\mathtt{0}, \mathtt{1}\}^*$ nochmals das Wort w schreibt (Kopiermaschine).

Anhang A

Lösungen

A.1 Lösungen zu den Übungen aus Kapitel [1.3]

A.1.1 Übung [1.1]

Gegeben ist das binäre Alphabet $\Sigma = \{0, 1\}$. Zeigen Sie, dass die Sprache

$$L = \{w \in \{0, 1\}^* \mid w = 0^n 1^n, n \in \mathbb{N}\}$$

entscheidbar ist.

Lösung:

Gemäß der Definition [1.9] ist eine Sprache L entscheidbar, falls es einen abbrechenden Algorithmus gibt, der für jedes $w \in \Sigma^*$ feststellt, ob $w \in L$ oder $w \notin L$.

Wir können folgendermaßen vorgehen, um festzustellen, ob $w \in L$ oder $w \notin L$ ist.

- Zunächst kann das Wort mit einer 1 beginnen, dann terminiert das Verfahren sofort, da das untersuchte Wort nicht die notwendige Struktur n 0en gefolgt von n 1en hat.
- Beginnt das Wort mit 0, dann vergleicht man, ob hinter dem (eventuell vorhandenen) 0er Block eine 1 kommt.
  - Kommt keine 1, wissen wir sofort, dass das Wort nicht in L liegt.
  - Folgt eine 1, wiederholen wir den Vergleichsvorgang.
- Das Verfahren muss feststellen
  - ob mehr 0en vorhanden sind als 1en, $\Longrightarrow w \notin L$,
  - ob mehr 1en vorhanden sind als 0en, $\Longrightarrow w \notin L$,

- ob hinter dem (eventuell vorhandenen) 1er Block noch weitere Zeichen folgen, $\Longrightarrow w \notin L$,
- ob gleich viele 0en und 1en vorhanden sind, $\Longrightarrow w \in L$.

Man kann sich das Verfahren klarmachen, wenn man sich überlegt, wie man ein Wort der Form

$$w = 0^n 1^m = \underbrace{000\cdots00}_{n}\underbrace{111\cdots11}_{m}, \quad n, m \in \mathbb{N}.$$

(mechanisch) verarbeitet, um festzustellen, ob auf n 0en gleich viele 1en folgen.

Step1: Lese das erste Zeichen links des Wortes. Falls es eine 1 ist, terminiere, $w \notin L$.

Step2: Falls das gelesene Zeichen eine 0 ist, streiche die 0.

Step3: Gehe nach rechts, bis keine 0 mehr gelesen wird. Folgt kein weiteres Zeichen auf die letzte 0 STOP, $w \notin L$. Falls auf die letzte 0 eine 1 folgt, streiche diese 1.

Step4: Gehe nach links bis zur ersten 0. Wiederhole die Schritte Step2 und Step3.

Step5: Wenn in diesem Schritt vor der gestrichenen 1 keine 0en mehr gelesen wird, gehe ein Zeichen nach rechts.

Step6; Wenn noch ein Zeichen folgt (0 oder 1), terminiere, $w \notin L$.

Step7: Wenn kein Zeichen mehr folgt, terminiere, $w \in L$.

Betrachte das Wort $w = 0001110$. Offensichtlich ist $w \notin L$, da dieser Bitstring nicht die Form $0^n 1^n$ hat.

$$\begin{aligned}
\underline{0}001110 &\xrightarrow{S1/S2} 001110\\
&\xrightarrow{S3} 00\not{1}\underline{1}110\\
&\xrightarrow{S4} \underline{0}0110\\
&\xrightarrow{S2} \not{0}\underline{0}0110\\
&\xrightarrow{S3} 0\not{1}\underline{1}10\\
&\xrightarrow{S4} \underline{0}10\\
&\xrightarrow{S2} \not{0}\underline{1}0\\
&\xrightarrow{S3} \underline{1}0\\
&\xrightarrow{S3} \not{1}\underline{0}\\
&\xrightarrow{S5} \underline{0} \qquad \text{Stop } w \notin L.
\end{aligned}$$

A.1.2 Übung [1.2]

Zeigen Sie, dass die Vereinigung von abzählbar unendlich vielen abzählbar unendlichen Mengen wieder abzählbar unendlich ist.

Lösung:

Gegeben sind abzählbar unendlich viele Mengen. Die erste dieser Menge nennen wir M_0. Da M_0 abzählbar ist, können wir die Elemente dieser Menge folgendermaßen notieren:

$$M_0 = \{m_{00}, m_{01}, m_{02}, m_{03}, m_{04}, \ldots\}.$$

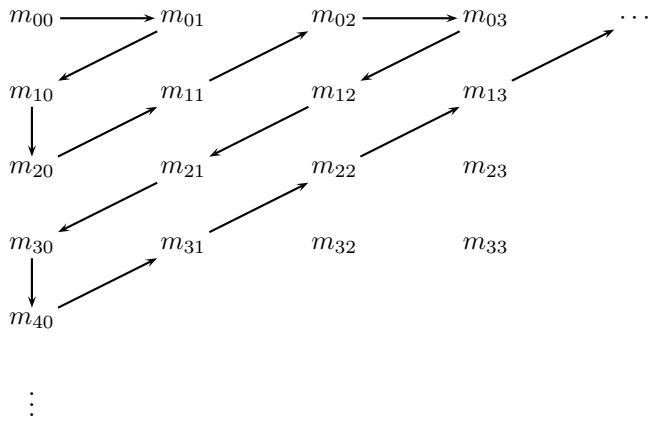
In der gleichen Weise kann man dies mit der Menge M_1 durchführen, *i.e.*

$$M_1 = \{m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, \ldots\}.$$

usw. Wir haben gemäß Voraussetzung abzählbar unendlich viele solcher Mengen $M_i, i = 0, 1, 2, \ldots$, die wir in dem folgenden Schema auflisten können:

$$\begin{aligned}
M_0 &= \{m_{00}, m_{01}, m_{02}, m_{03}, m_{04}, \ldots\} \\
M_1 &= \{m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, \ldots\} \\
M_2 &= \{m_{20}, m_{21}, m_{22}, m_{23}, m_{24}, \ldots\} \\
M_3 &= \{m_{30}, m_{31}, m_{32}, m_{33}, m_{24}, \ldots\} \\
\vdots &\qquad\quad \vdots
\end{aligned}$$

Wir können nun das erste CANTORsche Diagonalverfahren anwenden, um eine Aufzählung aller Elemente angeben zu können:



Damit ist:

$$
\begin{array}{rcl}
1 & \longleftrightarrow & m_{00} \\
2 & \longleftrightarrow & m_{01} \\
3 & \longleftrightarrow & m_{10} \\
4 & \longleftrightarrow & m_{20} \\
5 & \longleftrightarrow & m_{11} \\
6 & \longleftrightarrow & m_{02} \\
7 & \longleftrightarrow & m_{03} \\
8 & \longleftrightarrow & m_{12} \\
9 & \longleftrightarrow & m_{21} \\
10 & \longleftrightarrow & m_{30}
\end{array}
$$

A.1.3 Übung [1.3]

Überlegen Sie sich einen Algorithmus, der die Menge der natürlichen Zahlen in Binärdarstellung aufzählt.

Lösung:

Die Menge der natürlichen Zahlen ist (wir verwenden die Konvention, dass $0 \in \mathbb{N}$):

$$\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}.$$

Schreibt man die natürlichen im Binärsystem $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ auf, ergibt sich:

$$\begin{array}{rcl}
0 & \longleftrightarrow & \mathtt{0} \\
1 & \longleftrightarrow & \mathtt{1} \\
2 & \longleftrightarrow & \mathtt{10} \\
3 & \longleftrightarrow & \mathtt{11} \\
4 & \longleftrightarrow & \mathtt{100} \\
5 & \longleftrightarrow & \mathtt{101} \\
6 & \longleftrightarrow & \mathtt{110} \\
7 & \longleftrightarrow & \mathtt{111} \\
8 & \longleftrightarrow & \mathtt{1000} \\
9 & \longleftrightarrow & \mathtt{1001} \\
10 & \longleftrightarrow & \mathtt{1010} \\
11 & \longleftrightarrow & \mathtt{1011} \\
12 & \longleftrightarrow & \mathtt{1100} \\
13 & \longleftrightarrow & \mathtt{1101} \\
14 & \longleftrightarrow & \mathtt{1110} \\
15 & \longleftrightarrow & \mathtt{1111} \\
16 & \longleftrightarrow & \mathtt{10000}
\end{array}$$

usw. Damit können wir die natürlichen Zahlen schreiben im Binärsystem

$$\mathbb{N} = \{\mathtt{0}, \mathtt{1}, \mathtt{10}, \mathtt{11}, \mathtt{100}, \ldots\}.$$

Ein systematisches Verfahren — oder Algorithmus — das die natürlichen Zahlen in Binärdarstellung aufzählt, ermöglicht für eine Zahl n den Nachfolger zu erhalten. Sehen wir uns zu diesem Zweck als Beispiel die Zahl 39 an:

$$39 \longleftrightarrow \mathtt{100111}$$

Der Nachfolger ist 40, binär

$$40 \longleftrightarrow \mathtt{101000}$$

Um aus der Bitfolge `100111` die Folge `101000` zu erhalten, verfahren wir wie folgt. Beginnend mit dem kleinstwertigen Bit (das ist das Bit ganz rechts), ersetzen wir alle `1`en durch `0`en (von rechts nach links). Sobald die erste `0` auftritt, ersetzen wir diese durch eine `1`. Wenn keine `0` auftritt, fügt man am Anfang eine `1` hinzu.

A.1.4 Übung [1.4]

Zeigen Sie, dass für ein Alphabet Σ die Menge $\wp(\Sigma^*)$ überabzählbar unendlich ist.

Lösung:[1]

Betrachte das Alphabet Σ, gemäß Definition ist Σ eine endliche Menge.

Die Menge Σ^* ist die Menge aller Worte über dem Alphabet Σ. Ist beispielsweise $\Sigma = \{a, b\}$, dann ist

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, \ldots\}\,.$$

Diese Menge ist abzählbar unendlich, *i.e.* wir können die Wörter in dieser Menge abzählen, indem wir eine lexikographische Ordnung zugrundlegen, damit:

$$\begin{aligned} w_1 &= \lambda, \\ w_2 &= a, \\ w_3 &= b, \\ w_3 &= aa, \\ w_4 &= ab, \\ w_5 &= ba, \\ w_6 &= bb, \\ w_7 &= aaa, \\ w_8 &= aab, \\ &\vdots \quad \vdots \end{aligned}$$

Wir können also die Wörter aus Σ^* als unendlich lange, aber endliche Liste anordnen.

Wir nehmen an dass die Potenzmenge

$$\wp(\Sigma^*) = \{A \mid A \subseteq \Sigma^*\}$$

— *i.e.* die Menge aller Teilmengen von Σ^*, also die Menge aller Sprachen über dem Alphabet Σ — abzählbar ist. Dann kann jeder Teilmenge $M \subseteq \Sigma^*$ (also jedem $M \in \wp(\Sigma^*)$ in eindeutiger Weise eine natürliche Zahl $n \in \mathbb{N}$ zugeordnet werden. Man kan damit eine Liste abzählbar unendlicher Länge angeben, die an $n$ter Stelle die $n$te Teilmenge M_n enthält, wobei jedes Element von $\wp(\Sigma^*)$ in der Liste vorkommt.

Wir betrachten nun die (sortierten) Worte $w_1, w_2, \ldots$ aus der Menge Σ^* und stellen eine Tabelle auf, die aussagt, ob das Wort w_i in der Teilmenge M_j enthalten ist:

[1] Siehe auch [81]. Dies ist letztendlich CANTORs Beweis, dass *jede* Potenzmenge größer ist als die Menge selbst.

	w_1	w_2	w_3	w_4	w_5	w_6	$\cdots$
M_1	0	0	1	1	0	1	$\cdots$
M_2	1	0	1	0	0	1	$\cdots$
M_3	1	0	1	1	1	0	$\cdots$
M_4	0	1	1	0	1	0	$\cdots$
M_5	1	0	1	1	1	1	$\cdots$
M_6	0	0	1	1	0	0	$\cdots$
$\vdots$							

In jeder Zelle T_{ij} enthält die Tabelle einen Eintrag 1, wenn $w_i \in M_j$ (wenn also das Wort w_i in der Teilmenge M_j enthalten ist, sonst ist der Eintrag eine 0. Eine Zeile i in der obigen Tabelle gibt also die komplette Menge M_i an, also welches Wort $w_1, w_2, \ldots$ in der Menge enthalten ist.

Wir betrachten die Menge

$$M' = \{w_j \mid w_j \notin M_j\} \subseteq \Sigma^*, \quad \text{mit } j \in \mathbb{N}. \tag{A.1}$$

In der obigen Beispieltabelle — das ist nur eine Möglichkeit, die Mengen aufzulisten — ist

$$w_1 \in M', w_2 \in M', w_4 \in M', w_6 \in M', \qquad \text{usw,}$$

also

$$M' = \{w_1, w_2, w_4, w_6, \ldots\}.$$

Entsprechend unserer Annahme der Abzählbarkeit muss es in $n \in \mathbb{N}$ geben mit $M_n = M'$, denn alle Teilmengen von $Sigma^*$ sind in der Liste $M_1, M_2, \ldots$ enthalten. Die Menge M' ist jedoch gerade so konstruiert, dass sie sich in mindestens einem Element (dem Diagonalelement) von jeder Menge in der Liste unterscheidet. Dies erkennt man an dem Beispiel wie folgt:

$$\begin{array}{lll}
M' \neq M_1 & \text{weil} & w_1 \in M' \wedge w_1 \notin M_1, \\
M' \neq M_2 & \text{weil} & w_2 \in M' \wedge w_2 \notin M_2, \\
M' \neq M_3 & \text{weil} & w_3 \notin M' \wedge w_3 \in M_3, \\
M' \neq M_4 & \text{weil} & w_4 \in M' \wedge w_4 \notin M_4, \\
M' \neq M_5 & \text{weil} & w_5 \notin M' \wedge w_5 \in M_5, \\
M' \neq M_6 & \text{weil} & w_6 \in M' \wedge w_5 \notin M_6
\end{array}$$

usw.

Da es mindestens eine Menge gibt, die nicht in der Liste enthalten ist, muss die ursprüngliche Annahme falsch sein. Damit ist die Menge $\wp(\Sigma^*)$ nicht abzählbar, sondern überabzählbar.

A.1.5 Übung [1.5]

Sei Σ ein Alphabet. Beweisen Sie, dass nicht jede Funktion

$$f \,:\, \Sigma^* \longrightarrow \Sigma^*$$

berechenbar ist.

Lösung:

Wir zeigen die Behauptung über die folgenden beiden Aussagen:

(a) Die Menge aller Funktionen

$$f \,:\, \Sigma^* \longrightarrow \Sigma^*$$

ist überabzählbar.

(b) Die Menge aller berechenbaren Funktionen

$$f \,:\, \Sigma^* \longrightarrow \Sigma^*$$

ist abzählbar.

Dies impliziert, dass nicht jede Funktion berechenbar ist, da es mehr (viel mehr) Funktionen als berechenbare Funktionen gibt.

Um die erste Aussage zu zeigen führen wir einen Widerspruchsbeweis durch. Wir nehmen an, die Menge der Funktionen F der Form

$$f \,:\, \Sigma^* \longrightarrow \Sigma^*$$

ist abzählbar. Dann kann man die Menge F aufzählen, *i.e.* man kann alle Funktionen in F in einer unendlich langen, aber abzählbaren Liste anordnen:

$$F = \{f_1, f_2, f_3, \ldots\}\,.$$

wir wissen, die Menge Σ^* ist abzählbar unendlich, *i.e.* wir können die Wörter in dieser Menge abzählen, indem wir eine lexikographische Ordnung zugrundlegen. Für $\Sigma = \{a, b\}$ hat dies die Form:

$$\begin{aligned}
w_1 &= \lambda,\\
w_2 &= a,\\
w_3 &= b,\\
w_3 &= aa,\\
w_4 &= ab,\\
w_5 &= ba,\\
w_6 &= bb,\\
w_7 &= aaa,\\
w_8 &= aab,\\
&\vdots\quad\vdots
\end{aligned}$$

Wir können also die Wörter aus Σ^* als unendlich lange, aber endliche Liste anordnen:

$$\Sigma^* = \{w_1, w_2, w_3, \ldots\}.$$

Wenn nun die Annahme zutrifft, dann lassen sich alle Funktionen mitsamt ihren Funktionswerten in einer unendlich großen, aber abzählbaren Tabelle darstellen.

	w_1	w_2	w_3	w_4	w_5	w_6	$\cdots$
f_1	$f_1(w_1)$	$f_1(w_2)$	$f_1(w_3)$	$f_1(w_4)$	$f_1(w_5)$	$f_1(w_6)$	$\cdots$
f_2	$f_2(w_1)$	$f_2(w_2)$	$f_2(w_3)$	$f_2(w_4)$	$f_2(w_5)$	$f_2(w_6)$	$\cdots$
f_3	$f_3(w_1)$	$f_3(w_2)$	$f_3(w_3)$	$f_3(w_4)$	$f_3(w_5)$	$f_3(w_6)$	$\cdots$
f_4	$f_4(w_1)$	$f_4(w_2)$	$f_4(w_3)$	$f_4(w_4)$	$f_4(w_5)$	$f_4(w_6)$	$\cdots$
f_5	$f_5(w_1)$	$f_5(w_2)$	$f_5(w_3)$	$f_5(w_4)$	$f_5(w_5)$	$f_5(w_6)$	$\cdots$
f_6	$f_6(w_1)$	$f_6(w_2)$	$f_6(w_3)$	$f_6(w_4)$	$f_6(w_5)$	$f_6(w_6)$	$\cdots$
$\vdots$							

Wir definieren nun für Worte $u, v \in \Sigma^*, u \neq v$ eine Funktion

$$g \ : \ \Sigma^* \longrightarrow \Sigma^*$$

mit

$$\forall\, j \in \mathbb{N} \ : \ g(w_j) = \begin{cases} u & \text{falls } f_i(w_i) \neq u, \\ v & \text{falls } f_i(w_i) = u. \end{cases}$$

Die Funktion g unterscheidet sich also von allen Funktionen in der Tabelle in mindestens einem Funktionswert. Falls die Menge F abzählbar ist, muss ein $k \in \mathbb{N}$ existieren, so dass $g = f_k$. Dann muss auch gelten

$$g(w_k) = f_k(w_k).$$

Die Funktion g ist jedoch gerade so definiert, dass

$$g(w_k) \neq f_k(w_k).$$

Dies ist ein Widerspruch, daher ist die ursprünglichen Annahme falsch, dass die Menge der Funktionen abzählbar ist.

Die Menge der berechenbaren Funktionen F_b ist abzählbar. Dies kann man sich folgendermaßen klarmachen (dies ist kein Beweis, dies ist eine informelle Überlegung). Wenn die Funktion $f \in F_b$ berechenbar ist, dann gibt es einen Text endlicher Länge über einem Alphabet Σ zur Beschreibung eines die Funktion f berechnenden Algorithmus. Da die Menge aller Texte Σ^* abzählbar ist, ist die Menge aller berechenbarer Funktionen abzählbar.

A.1.6 Übung [1.6]

Betrachte das Alphabet $\Sigma = \{a, b\}$. Betrachte die Worte $w_1 = a^2ba^3b^2$ und $w_2 = bab^2$. Bestimmen Sie:

(a) $w_1w_2, \quad | w_1w_2 |$

(b) $w_2w_1, \quad | w_2w_1 |$

(c) $w_2^2, \quad | w_2^2 |$

Lösung:

(a)
$$w_1 \circ w_2 = (a^2ba^3b^2)(bab^2) = a^2ba^3b^3ab^2, \qquad | w_1 \circ w_2 | = 12.$$

(b)
$$w_2 \circ w_1 = (bab^2)(a^2ba^3b^2) = bab^2a^2ba^3b^2, \qquad | w_2 \circ w_3 | = 12.$$

(c)
$$w_2^2 = w_2 \circ w_2 = (bab^2)(bab^2) = bab^3ab^2, \qquad | w_2^2 | = 8.$$

A.1.7 Übung [1.7]

Sei $\Sigma = \{a, b\}$. Beschreiben Sie verbal die folgenden Sprachen über dem Alphabet Σ.

(a) $L_1 = \{(ab)^n \mid n > 0\}$

(b) $L_2 = \{a^r ba^s ba^t \mid r, s, t \geq 0\}$

(c) $L_3 = \{a^2 b^m a^3 \mid m \in \mathbb{N}, m > 0\}$

Lösung:

(a) L_1 besteht aus allen Worten der Form $abababab\ldots$, d.h. die Worte beginnen mit a, wechseln sich immer mit b ab und enden mit b. Oder: Die Worte bestehen aus einer Anzahl von ab–Paaren.

(b) L_2 besteht aus allen Wörtern mit genau zwei b.

(c) L_3 besteht aus allen Wörtern die mit aa beginnen und mit aaa enden, dazwischen liegen Teilstrings, die aus einem oder mehreren b bestehen.

A.1.8 Übung [1.8]

Seien $L_1 = \{a, ab, bb\}$ und $L_2 = \{b^2, aba\}$ zwei Sprachen über dem Alphabet $\Sigma = \{a, b\}$. Geben Sie die folgenden Sprachen an:

$$L_1L_2, L_2L_1, L_2^0, L_2^2, L_2^{-2}, (L_1L_2)^2, L_1^2L_2^2.$$

Lösung:

Es gilt:

$$\begin{aligned}
L_1L_2 &= \{a, ab, bb\} \circ \{b^2, aba\} = \{ab^2, a^2ba, ab^3, ababa, b^4, b^aba\} \\
L_2L_1 &= \{b^2, aba\} \circ \{a, ab, bb\} = \{b^2a, b^2ab, b^4, aba^2, aba^2b, abab^2\} \\
L_2^0 &= \{\lambda\} \\
(L_2)^2 &= L_2 \circ L_2 \\
&= \{b^2, aba\} \circ \{b^2, aba\} \\
&= \{b^4, b^2aba, abab^2, aba^2ba\} \\
L_2^{-2} &\quad \text{ist nicht definiert für negative Potenzen} \\
(L_1L_2)^2 &= (L_1L_2) \circ (L_1L_2) \\
&= \{ab^2, a^2ba, ab^3, ababa, b^4, b^aba\} \circ \{ab^2, a^2ba, ab^3, ababa, b^4, b^aba\} \\
&= \{ab^2ab^2, ab^2a^2ba, \ldots\} \\
L_1^2L_2^2 &= (\{a, ab, bb\})^2 \circ \left(\{b^2, aba\}\right)^2 \\
&= \{a^2, a^bab^2, aba, (ab)^2, ab^3, b^2a, b^2ab, b^4\} \circ \{b^4, b^2aba, abab^2, (aba)^2\} \\
&= \{a^2b^4, a^2b^2aba, a^3bab^2, a^2(aba)^2, \ldots\}
\end{aligned}$$

A.1.9 Übung [1.9]

Sei $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ das binäre Alphabet. Betrachte die Sprachen

$$L_1 = \{w \in \Sigma^* \mid w = w'\mathtt{11}, w' \in \Sigma^*\},$$
$$L_2 = \{w \in \Sigma^* \mid w = \mathtt{11}w'.w' \in \Sigma^*\}.$$

Geben Sie die folgenden Mengen an:

(a) $L_1 \cup L_2$

(b) $L_1 \cap L_2$

(c) $\overline{L_1}$

(d) $L_1 \setminus L_2$

(e) $L_2 \setminus L_1$

(f) $\overline{(L_1 \cap L_2)}$

(g) $\overline{L_1} \cap \overline{L_2}$

Lösung:

Die Sprache L_1 besteht aus allen Bitstrings, die mit $\mathtt{11}$ enden, L_2 sind die Bitstrings, die mit $\mathtt{11}$ beginnen.

(a) Dann ist $L_1 \cup L_1$ die Menge aller Bitstrings, die mit $\mathtt{11}$ beginnen oder enden (oder beides).

(b) Dann ist $L_1 \cap L_1$ die Menge aller Bitstrings, die mit $\mathtt{11}$ beginnen *und* mit $\mathtt{11}$ enden.

(c) Die Komplementmenge $\overline{L}$ ist definiert

$$\overline{L_1} = \{w \in \Sigma^* \mid w \notin L_1\}.$$

Das sind (informell) alle Bitstrings, die nicht mit $\mathtt{11}$ enden, beispielsweise:

$$\overline{L_1} = \{\lambda, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{000}, \mathtt{001}, \mathtt{010}, \mathtt{100}, \mathtt{110}, \mathtt{101}, \ldots\}.$$

(d) Die Menge $L_1 \setminus L_2$ ist definiert durch

$$L_1 \setminus L_2 = \{w \in \Sigma^* \mid w \in L_1 \wedge w \notin L_2\},$$

i.e. dies sind alle Strings, die in L_1 liegen und nicht in L_2. Informell sind dies alle Bitstrings, die mit $\mathtt{11}$ enden und nicht mit $\mathtt{11}$ beginnen.

$$L_1 = \{\mathtt{11}, \mathtt{011}, \mathtt{111}, \mathtt{0011}, \mathtt{0111}, \mathtt{1011}, \mathtt{1111}, \cdots\}$$
$$L_2 = \{\mathtt{11}, \mathtt{110}, \mathtt{111}, \mathtt{1100}, \mathtt{1101}, \mathtt{1110}, \mathtt{1111}, \cdots\}.$$

Dann ist

$$L_1 \setminus L_2 = \{\mathtt{011}, \mathtt{0011}, \mathtt{0111}, \mathtt{1011}, \cdots\}.$$

(e) Analog ist

$$L_2 \setminus L_1 = \{w \in \Sigma^* \mid w \in L_2 \wedge w \notin L_1\},$$

i.e. dies sind alle Bitstring, die mit `11` beginnen und nicht mit `11` enden, beispielsweise

$$L_2 \setminus L_1 = \{\mathtt{110}, \mathtt{1100}, \mathtt{1101}, \mathtt{1110}, \mathtt{11000}, \cdots\}.$$

(f) Informell können wir die Menge $\overline{(L_1 \cap L_2)}$ beschreiben als alle Strings über dem Alphabet $\{\mathtt{0}, \mathtt{1}\}$, die entweder nicht mit `11` beginnen oder nicht mit `11` enden (oder beides). Man beachte es gilt die DeMorganschen Regel

$$\overline{(L_1 \cap L_2)} = \overline{L_1} \cup \overline{L_2}.$$

Mit

$$\overline{L_1} = \{\lambda, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{000}, \mathtt{001}, \mathtt{010}, \mathtt{100}, \mathtt{110}, \mathtt{101}, \ldots\}$$

und

$$\overline{L_2} = \{\lambda, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{000}, \mathtt{001}, \mathtt{010}, \mathtt{100}, \mathtt{011}, \mathtt{101}, \ldots\}$$

erhalten wir

$$\overline{(L_1 \cap L_2)} = \{\lambda, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{000}, \mathtt{001}, \mathtt{010}, \mathtt{100}, \mathtt{101}, \mathtt{011}, \mathtt{110} \cdots\}.$$

(g) Informell können wir die Menge $\overline{L_1} \cap \overline{L_2}$ beschreiben als alle Strings über dem Alphabet $\{\mathtt{0}, \mathtt{1}\}$, die weder mit `11` beginnen noch mit `11` enden,

$$\overline{L_1} \cap \overline{L_2} = \{\lambda, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{000}, \mathtt{100}, \mathtt{010}, \mathtt{001}, \mathtt{101} \cdots\}.$$

A.1.10 Übung [1.10]

Betrachte das Alphabet $\Sigma = \{a, b\}$ und die Sprache

$$L = \{w \in \Sigma^* \mid w = uu^R, u \in \Sigma^*, u^R \text{ ist das Spiegelbild von } u\}.$$

Betrachten Sie den folgenden Algorithmus. der für beliebige Worte $w \in \Sigma^*$ als Input in der folgenden Weise arbeitet:

```
while( |w| >= 2)
    {
     if(erstes Zeichen von w ungleich letztes Zeichen von w)
       entferne erstes und letztes Zeichen von w
     else
      vertausche erstes und letztes Zeichen von w
    }
if( |w| = 0)
   return nein
else
   return ja
```

(a) Ändern Sie diesen Algorithmus so ab, dass daraus ein Entscheidungsverfahren für L entsteht.

(b) Ist die Sprache L abzählbar? Wenn ja, beschreiben Sie, wie die Wörter aus L durchnummeriert werden können.

Lösung:

(a) Ein Entscheidungsverfahren für eine Sprache ist gemäß Definition [1.9] ein systematische Verfahren, welches für jedes Wort über dem zugrundeliegenden Alphabet fetstellt, ob das Wort in der Sprache L ist oder nicht. Der folgende Algorithmus in C–Pseudocode prüft, ob das erste und letzte Zeichen des Wortes w gleich ist. Wenn dies für alle Wörter der Fall ist, dann ist die zweite Hälfte des Wortes ein Spiegelbild der ersten Hälfte. Dann gibt der Algorithmus ein *ja* zurück, andernfalls ein *nein*.

```
while( |w| >= 2)
    {
     if(erstes Zeichen von w gleich letztes Zeichen von w)
         entferne erstes und letztes Zeichen von w
     else
         return nein
  }
if( |w| = 0)
   return ja
else
   return nein
```

(b) Wir wissen, da $\Sigma = \{a, b\}$ ein endliches Alphabet ist, dass die Menge aller Worte $\{a, b\}^*$ über diesem Alphabet abzählbar unendlich ist. Wir können die Worte in $\{a, b\}^*$ beispielsweise lexikographisch ordnen.

$$\Sigma^* = \{\lambda, a, b, \underline{aa}, ab, ba, \underline{bb}, \ldots$$
$$\ldots \underline{aaaa}, aaab, aaba, aabb, abaa, abab, \underline{abba}, abbb, baaa, \underline{baab}, baba, babb, bbaa, bbab, bbba, \underline{bbbb}$$
$$\ldots\}$$

Die unterstrichenen Wörter in dieser Liste sind $w \in L$. Wir erhalten eine Nummerierung der Wörter der Sprache L, indem wir dem Wort ww^R einfach die Nummer des Wortes $w \in \Sigma^*$ zuordnen.

A.2 Lösungen zu den Übungen aus Kapitel [2.4]

A.2.1 Übung [2.11]

Gegeben ist die Grammatik $G = (N, T, P, S)$ mit

- $N = \{S, A, B\}$
- $T = \{0\}$
- P mit

$$\begin{array}{rcl} S & \longrightarrow & \lambda \\ S & \longrightarrow & ABA \\ AB & \longrightarrow & 00 \\ 0A & \longrightarrow & 000A \\ A & \longrightarrow & 0. \end{array}$$

- S ist Startvariable.

(a) Von welchem Typ ist diese Grammatik?

(b) Geben Sie eine Ableitung für das Wort $w = 00000$ in G an.

(c) Beschreiben Sie die Sprache $L(G)$.

(d) Geben Sie eine zu G äquivalente Grammatik G' an.

Lösung:

(a) Die Form der Grammatik, wie sie oben gegeben ist, ist eine Typ 0 Grammatik, denn es gibt keine Einschränkung an die Regeln.

(b) Das Wort $w = 00000$ kann folgendermaßen abgeleitet werden:

$$\begin{array}{rcl} S & \Longrightarrow & ABA \\ & \Longrightarrow & 00A \\ & \Longrightarrow & 0000A \\ & \Longrightarrow & 00000. \end{array}$$

(c) Die Sprache L, die die Grammatik G erzeugt, ist

$$L = \{\lambda\} \cup \{w \in \{0\}^* \mid w = 0^{2n+1}, n \geq 1\}.$$

Die Ableitung von Wörtern beginnt mit der Regel $S \longrightarrow ABA$. Dann kann nur die kontextsensitive Regel $AB \longrightarrow 00$ angewendet werden, es ergibt sich stets der String $00A$. Mit jeder Anwendung der vierten Regel werden zwei 0en hinzugefügt, so dass nach n Schritten der String $0^{2n}A$ entsteht. Eine andere Möglichkeit gibt es nicht. Wird schließlich die letzte Regel angewendet, entsteht ein String der Form 0^{2n+1}.

(d) Die Sprache L kann auch durch eine rechtslineare Grammatik erzeugt werden, mit

$$G' = (N', T, P', S)$$

mit

- $N = \{S, A, B\}$
- $T = \{0\}$
- P mit

$$\begin{array}{lcl} S & \longrightarrow & \lambda \\ S & \longrightarrow & 0A \\ A & \longrightarrow & 0B \\ B & \longrightarrow & 0A \mid 0 \end{array}$$

- S ist Startvariable.

A.2.2 Übung [2.12]

Gegeben sind die Sprachen

$$L_1 = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid\mid w \mid_0 \mod 2 = 0 \text{ und } \mid w \mid_1 = 1\}$$
$$L_2 = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w \neq u\mathtt{00}v, w \neq u\mathtt{11}v\}$$

Geben Sie jeweils eine rechtslineare Grammatik G_i an, so dass $L_i = L(G_i)$.

Lösung:

1. Die Sprache L_1 besteht aus allen Wörtern über dem Alphabet $\Sigma = \{\mathtt{0},\mathtt{1}\}$, die eine gerade Anzal von `0`en und eine ungerade Anzahl von `1`en haben. Eine rechtslineare Grammatik, die diese Sprache erzeugt ist:

   $$G_1 = (N, T, P, S)$$

   mit den vier Variablen

   $$N = \{S, A, B, C\},$$

   dem Terminalalphabet

   $$T = \{\mathtt{0},\mathtt{1}\},$$

   der Startvariablen S und der Menge der Produktionen P:

   $$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}B \mid \mathtt{1}A \mid \mathtt{1} \\ A & \longrightarrow & \mathtt{0}C \mid \mathtt{1}S \\ B & \longrightarrow & \mathtt{0}S \mid \mathtt{1}C \\ C & \longrightarrow & \mathtt{0}A \mid \mathtt{1}B \mid \mathtt{0}. \end{array}$$

2. Die Sprache L_2 besteht aus allen Wörtern über dem Alphabet $\Sigma = \{\mathtt{0},\mathtt{1}\}$, die weder den Teilstring `00` noch den String `1` enthalten. Eine rechtslineare Grammatik, die diese Sprache generiert ist:

   $$G_2 = (N, T, P, S)$$

   mit den drei Variablen

   $$N = \{S, A, B\},$$

   dem Terminalalphabet

   $$T = \{\mathtt{0},\mathtt{1}\},$$

   der Startvariablen S und der Menge der Produktionen P:

   $$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}A \mid \mathtt{1}B \mid \lambda \\ A & \longrightarrow & \mathtt{1}B \mid \mathtt{1} \\ B & \longrightarrow & \mathtt{0}A \mid \mathtt{0} \end{array}$$

A.2.3 Übung [2.13]

Geben Sie eine rechtslineare Grammatik G an mit $L = L(G)$, wobei

$$L = \{w \in \{\mathtt{0},\mathtt{1}\}^+ \mid \forall\, u, v \in \{\mathtt{0},\mathtt{1}\}, w \neq u\mathtt{1111}v\}.$$

Wörter in L sind also mindestens ein Zeichen lang und dürfen den Teilstring `1111` nicht enthalten. Produzieren Sie das Testwort `011100110001110`.

Lösung:

Die Grammatik ist:

$$G = (N, T, P, S)$$

mit $N = \{S, A, B, C\}, T = \{\mathtt{0},\mathtt{1}\}$ und den Produktionen P

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}S \mid \mathtt{1}A \mid \mathtt{1} \mid \mathtt{0} \\ A & \longrightarrow & \mathtt{0}S \mid \mathtt{1}B \mid \mathtt{1} \mid \mathtt{0} \\ B & \longrightarrow & \mathtt{0}S \mid \mathtt{1}C \mid \mathtt{1} \mid \mathtt{0} \\ C & \longrightarrow & \mathtt{0}S \mid \mathtt{0} \end{array}$$

Das Testwort `011100110001110` wird folgendermaßen abgeleitet:

$$\begin{array}{rcl} S & \Longrightarrow & \mathtt{0}S \\ & \Longrightarrow & \mathtt{01}A \\ & \Longrightarrow & \mathtt{011}B \\ & \Longrightarrow & \mathtt{0111}C \\ & \Longrightarrow & \mathtt{01110}S \\ & \Longrightarrow & \mathtt{011100}S \\ & \Longrightarrow & \mathtt{0111001}A \\ & \Longrightarrow & \mathtt{01110011}B \\ & \Longrightarrow & \mathtt{011100110}S \\ & \Longrightarrow & \mathtt{0111001100}S \\ & \Longrightarrow & \mathtt{01110011000}S \\ & \Longrightarrow & \mathtt{011100110001}A \\ & \Longrightarrow & \mathtt{0111001100011}B \\ & \Longrightarrow & \mathtt{01110011000111}C \\ & \Longrightarrow & \mathtt{011100110001110} \end{array}$$

A.2.4 Übung [2.14]

Gegeben sind zwei Grammatiken mit den Regelmengen P_1 bzw. P_2, die in der erweiterten Backus–Naur Form folgendermaßen aussehen:

$$P_1 : < S > ::= a[< S >]a \mid b$$
$$P_2 : < S > ::= a\{< S >\}a \mid b$$

(a) Geben Sie ein Wort an, das von der zweiten Grammatik erzeugt wird, aber nicht von der ersten Grammatik.

(b) Stellen Sie die EBNF–Regeln als Grammatikregel in theorieorientierter Form dar.

(c) Stellen Sie die EBNF–Regeln als Syntax–Diagramm dar.

Lösung:

A.2.5 Übung [2.15]

Geben Sie eine kontextfreie Grammatik an, die die Gliederung von Breifen beschreibt, die nach dem folgenden Prinzip strukturiert sind:

- Ein Brief besteht aus Einleitung, Haupttext und Schluss.
- Die Einleitung besteht aus Datum, Referenzzeichen, Empfängeranschrift und Betreff, wobei das Referenzzeichen auch fehlen kann.
- Der Haupttext besteht aus einer Anrede und einer Liste von Absätzen. Die Absätze bestehen jeseils aus einer Liste von Sätzen.
- Der Schluss besteht aus Schlussformel, Verfasser und Funktion des Verfassers.

Weiter als angegeben soll die Gliederung nicht in die Tiefe gehen. Die bei dieser Gliederung nicht unterteilbaren Bestandteile sollen als terminale Zeichen dargestellt werden. Alle angesprochenen Listen sollen nicht leer sein.

Lösung:

Die folgenden Grammatikregeln beschreiben den Aufbau von Briefen. Die Variablen sind hierbei in Großbuchstaben gesetzt, die Terminale sind in kleinen Buchstaben geschrieben.

BRIEF	$\longrightarrow$	EINLEITUNG HAUPTTEIL SCHLUSS
EINLEITUNG	$\longrightarrow$	datum refenzzeichen empfaengeranschrift betreff
HAUPTTEIL	$\longrightarrow$	anrede ABSATZLISTE
ABSATZLISTE	$\longrightarrow$	ABSATZ | ABSATZLISTE ABSATZ
ABSATZ	$\longrightarrow$	satz | ABSATZ satz
SCHLUSS	$\longrightarrow$	schlussformel verfasser funktion

A.2.6 Übung [2.16]

Gegeben ist die Grammatik

$$G = (N, T, P, S)$$

mit $N = \{S\}, T = \{o, u, r, l\}$ und den Produktionen P:

$$\begin{array}{rcl} S & \longrightarrow & oSu \\ S & \longrightarrow & rSl \\ S & \longrightarrow & ou \\ S & \longrightarrow & rl \\ or & \longrightarrow & ro \\ ou & \longrightarrow & uo \\ ol & \longrightarrow & lo \\ ro & \longrightarrow & or \\ ru & \longrightarrow & ur \\ rl & \longrightarrow & lr \\ uo & \longrightarrow & ou \\ ur & \longrightarrow & ru \\ ol & \longrightarrow & lu \\ lo & \longrightarrow & ol \\ lr & \longrightarrow & rl \\ lu & \longrightarrow & ul \end{array}$$

Die Terminalen stehen für Bewegungen eines Cursors auf dem Bildschirm eines Computers, der in die vier Richtungen o – oben, u – unten, r – rechts, und l links bewegt werden kann. Es versteht sich, dass pro terminalem Zeichen eine feste Längeneinheit zurückgelegt wird.

(a) Geben Sie einen möglichst spezielle Typ für die Grammtik G an.

(b) Skizzieren Sie den Weg, der durch das Wort *oruullor* zurückgelegt wird.

(c) Geben Sie eine Ahbleitung des Wortes *oruullor* in der Grammatik an.

(d) Wie sieht die Sprache aus, die diese Grammatik erzeugt?

Lösung:

Die oben aufgeführte Grammatik unterliegt keinerlei Einschränkungen, sie ist vom Typ 0.

A.2.7 Übung [2.17]

Geben Sie eine kontextfreie Grammatik G an, für die gilt $L = L(G)$ und

$$L = \{w \in \{0,1\}^* \mid w = 0^i 1^j, i,j \geq 0, i < j\}.$$

Lösung:

Es ist zunächst hilfreich, ein paar Worte dieser Sprache aufzulisten.

$$\begin{aligned} L &= \{w \in \{0,1\}^* \mid w = 0^i 1^j, i,j \geq 0, i < j\} \\ &= \{1, 011, 0111, 01111, \ldots, 00111, 001111 \ldots\}. \end{aligned}$$

Worte dieser Sprache bestehen also aus einem 0er Block, gefolgt von einem 1er Block. Dabei hat der 1er Block mindestens ein Zeichen mehr als der 0er Block.

Die Erzeugung von 01 Paaren mit gleicher Anzahl erhalten wir durch eine Regel der Form

$$S \longrightarrow 0S1.$$

Durch wiederholte Ausführung dieser Produktion ergeben sich Strings der Form $0^k S 1^k$.

Um zu erreichen, dass der finale Terminalzeichenstring mindestens eine 1 mehr enthält als 0en, fügen wir eine Regel

$$S \longrightarrow S1$$

hinzu, was natürlich ebenfalls möglich ist, ist die Regel

$$S \longrightarrow 1.$$

Damit ist:

$$G = (N, T, P, S),$$

mit $N = \{s\}$, $T = \{0,1\}$, Startvariable S und den drei Produktionen P:

$$\begin{aligned} S &\longrightarrow 0S1 \\ S &\longrightarrow S1 \\ S &\longrightarrow 1. \end{aligned}$$

A.2.8 Übung [2.18]

Gegeben sind die beiden folgenden Sprachen L_1 und L_2

$$L_1 = \{a^{2k+1}b^{2k}, k \geq 0\},$$
$$L_2 = \{(ab)c^{2k}, k > 0\}.$$

Geben Sie kontextfreie Grammatiken G_1 und G_2 an mit $L_1 = L(G_1)$ und $L_2 = L(G_2)$. Geben Sie jeweils Ableitungen zweier Beispielstrings an.

Lösung:

Es gilt:

$$G_1 = (\{S\}, \{a, b\}, P_1, S)$$

mit

$$P_1 = \{S \longrightarrow aaSbb \mid a\}.$$

Die kontextfreie Grammatik G_2 ist:

$$G_2 = (\{S\}, \{a, b, c\}, P_2, S)$$

mit den Produktionen

$$P_2 = \{S \longrightarrow abScc \mid abcc\}.$$

A.2.9 Übung [2.19]

Betrachte das Alphabet $\Sigma = \{a, b\}$ und die Sprache L über Σ, die aus allen Worten besteht, die genau ein b enthalten, *i.e.*

$$L = \{b, a^r b, ba^s, a^r ba^s; \quad r, s > 0\}.$$

Zeigen Sie, dass L eine reguläre Sprache ist.

Lösung:

Um zu zeigen, dass die Sprache L regulär ist, müssen wir eine rechtslineare Grammatik G angeben, mit $L = L(G)$. Es ist:

$$G = (N, T, P, S),$$

mit der Menge der Variablen

$$N = \{S, A, B\},$$

dem Terminalalphabet $T = \{a, b\}$, der Startvariablen S und den Produktionen:

$$\begin{aligned} S &\longrightarrow b \mid bA \mid aB \\ A &\longrightarrow a \mid aA \\ B &\longrightarrow aB \mid bA \mid b. \end{aligned}$$

Das Zeichen b wird durch diese Regeln genau einmal erzeugt.

- Strings der Form $a^n b$:

$$S \Longrightarrow aB \Longrightarrow aaB \longrightarrow \cdots \Longrightarrow a^n B \Longrightarrow a^n b.$$

- Strings der Form ba^n:

$$S \Longrightarrow bA \Longrightarrow baA \longrightarrow \cdots \Longrightarrow ba^{n-1}A \Longrightarrow ba^n.$$

- Strings der Form $a^m ba^n$:

$$S \Longrightarrow aB \Longrightarrow a^m B \longrightarrow a^m bA \Longrightarrow a^m baA \Longrightarrow a^m ba^n.$$

A.2.10 Übung [2.20]

Betrachten Sie die folgenden Aussagen über Sprachen über einem Alphabet Σ. Argumentieren Sie, warum diese wahr oder falsch sind.

(a) Wenn die Sprachen L_1 und L_2 regulär sind, dann ist jede Sprache L mit

$$L_1 \subseteq L \subseteq L_2$$

regulär.

(b) Sei L_1 eine reguläre Sprache und L_2 eine nicht–reguläre Sprache. Dann ist $L_1 \cap L_2$ regulär.

(c) Der Durchschnitt zweier nicht–regulärer Sprachen ist stets nicht regulär.

(d) Die Menge der kontextfreien Sprachen ist gegen Komplementbildung abgeschlossen.

Hinweis:
Für ein Alphabet Σ sind die Sprachen $\emptyset$ und Σ^* durch `DFA`, `NFA` oder rechtslineare Grammatiken darstellbar und somit regulär.

Lösung:

(a) Die Aussage:

> Wenn die Sprachen L_1 und L_2 regulär sind, dann ist jede Sprache L mit $L_1 \subseteq L \subseteq L_2$ regulär

ist falsch.
Erklärung:
Wir geben ein Gegenbeispiel an: Die Sprache $L_1 = \emptyset$ ist regulär, die Sprache Σ^* ist regulär. Nun wissen wir, dass die Sprache

$$L = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w = \mathtt{0}^n\mathtt{1}^n, n \geq 0\}$$

nicht regulär ist. Dann ist aber

$$\emptyset \subseteq L \subseteq \{\mathtt{0},\mathtt{1}\}^*.$$

Damit ist die Aussage falsch.

(b) Die Aussage: Sei L_1 eine reguläre Sprache und L_2 eine nicht–reguläre Sprache. Dann ist $L_1 \cap L_2$ regulär, ist falsch.
Erklärung:
Wir wissen, dass die Sprache $\Sigma^* = \{\mathtt{0},\mathtt{1}\}^*$ regulär ist und die Sprache

$$L = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w = \mathtt{0}^n\mathtt{1}^n, n \geq 0\}$$

nicht regulär ist. Dann ist die Schnittmenge

$$\Sigma^* \cap L = L,$$

was wiederum eine nicht reguläre Sprache ist. Damit ist ein Gegenbeispiel gezeigt, die Aussage ist also falsch.

(c) Die Aussage

> Der Durchschnitt zweier nicht–regulärer Sprachen ist stets nicht regulär.

ist falsch.
Erklärung:
Seien

$$L_1 = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w = \mathtt{0}^n\mathtt{1}^n, n \geq 0\}$$

und

$$L_2 = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w = \mathtt{1}^n\mathtt{0}^n, n \geq 0\}$$

zwei nicht–reguläre Sprachen. Dann ist

$$L_1 \cap L_2 = \emptyset.$$

Die Sprache $L = \emptyset$ ist regulär. Daher ist die Aussage falsch.

(d) Die Aussage

> Die Menge der kontextfreien Sprachen ist gegen Komplementbildung abgeschlossen.

ist falsch.
Erklärung:
Die Sprachen

$$L_1 = \{a^m b^n c^n, m, n \geq 0\} \quad \text{und} \quad L_2 = \{a^n b^n c^m, m, n \geq 0\}$$

sind kontextfrei, der Durchschnitt

$$L_1 \cap L_2 = \{a^n b^n c^n, n \in \mathbb{N}\}$$

jedoch nicht, $L_1 \cap L_2$ ist eine kontextsensitive Sprache.

Wir nehmen an, die kontextfreien Sprachen sind abgeschlossen unter Komplementbildung. Sind L_1, L_2 kontextfreie Sprachen, dann sind auch $\overline{L_1}$ und $\overline{L_2}$ kontextfrei. Wenn L_1, L_2 kontextfrei sind, dann ist auch die Vereinigung $L_1 \cup L_2$ kontextfrei. Dann ist auch

$$\overline{L_1} \cup \overline{L_2}$$

kontextfrei. Dann muss aber nach der Regel von DeMorgan auch

$$\overline{(\overline{L_1} \cup \overline{L_2})} = \overline{(\overline{L_1})} \cap \overline{(\overline{L_2})} = L_1 \cap L_2$$

kontextfrei sein, was im Widerspruch dazu steht, dass $L_1 \cap L_2$ nicht kontextfrei ist.

A.3 Lösungen zu den Übungen aus Kapitel [3.14]

A.3.1 Übung [3.21]

Geben Sie zu den folgenden regulären Sprachen über dem Alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ jeweils eine reguläre Grammatik an, die die Sprache erzeugt.

(a) $L_1 = \{w \in \Sigma^* \mid w$ beginnt mit einer ungeraden Anzahl 0en, gefolgt von einer geraden Anzahl 1en$\}$.

(b) $L_2 = \{w \in \Sigma^* \mid w$ enthält mindestens zwei 0en und höchstens eine 1$\}$.

(c) $L_3 = \{w \in \Sigma^* \mid w$ enthält nicht das Teilwort 110$\}$.

Lösung:

(a) Die gesuchte Grammatik ist das Viertupel

$$G_1 = (N, T, P, S)$$

mit der Menge der Variablen

$$V = \{S, X, Y, Z\},$$

der Menge der Terminalzeichen

$$T = \{\mathtt{0}, \mathtt{1}\},$$

der Startvariablen S und den Produktionen

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}X \mid \mathtt{0} \\ X & \longrightarrow & \mathtt{0}S \mid \mathtt{1}Y \\ Y & \longrightarrow & \mathtt{1}Z \mid \mathtt{1} \\ Z & \longrightarrow & \mathtt{1}Y. \end{array}$$

(b) Die gesuchte Grammatik ist das Viertupel

$$G_2 = (N, T, P, S)$$

mit der Menge der Variablen

$$V = \{A, B, C, D, E\},$$

der Menge der Terminalzeichen

$$T = \{\mathtt{0}, \mathtt{1}\},$$

der Startvariablen A und den Produktionen

$$\begin{array}{lcl} A & \longrightarrow & \mathtt{0}B \mid \mathtt{1}C \\ C & \longrightarrow & \mathtt{0}D \\ D & \longrightarrow & \mathtt{0}D \mid \mathtt{0} \\ B & \longrightarrow & \mathtt{0}B \mid \mathtt{1}D \mid \mathtt{0}E \mid \mathtt{0} \\ E & \longrightarrow & \mathtt{1}. \end{array}$$

Beispiel
Diese Grammatik erzeugt alle Bitstrings der Form

$$\mathtt{0}^{n+2}, \mathtt{10}^{n+2}, \mathtt{0}^{n+2}\mathtt{1} \text{ und } \mathtt{0}^k\mathtt{10}^l, k+l \geq 2, n \geq 0, k, l, n \in \mathbb{N}_0,$$

also alle Bitstring mit mindestens zwei `0`en und höchstens eine (das sind keine oder genau eine) `1`.

(a) String $\mathtt{00000\ldots0}$. Dazu

$$\begin{array}{rcll} A & \Longrightarrow & \mathtt{0}B & \text{Regel: } A \to \mathtt{0}B \\ & \Longrightarrow & \mathtt{00}B & \text{Regel: } B \to \mathtt{0}B \\ & \Longrightarrow & \mathtt{000}B & \text{Regel: } B \to \mathtt{0}B \\ & \vdots & & \\ & \Longrightarrow & & \\ & \Longrightarrow & \mathtt{000\ldots0} & \text{Regel: } B \to \mathtt{0} \end{array}$$

(b) String $\mathtt{00000\ldots1}$. Dazu

$$\begin{array}{rcll} A & \Longrightarrow & \mathtt{0}B & \text{Regel: } A \to \mathtt{0}B \\ & \Longrightarrow & \mathtt{00}B & \text{Regel: } B \to \mathtt{0}B \\ & \Longrightarrow & \mathtt{000}B & \text{Regel: } B \to \mathtt{0}B \\ & \vdots & & \\ & \Longrightarrow & & \text{usw.} \\ & \Longrightarrow & \mathtt{000\ldots0E} & \text{Regel: } B \to \mathtt{0}E \\ & \Longrightarrow & \mathtt{000\ldots01} & \text{Regel: } E \to \mathtt{1} \end{array}$$

(c) String $\mathtt{100000\ldots0}$. Dazu

$$\begin{array}{rcll} A & \Longrightarrow & \mathtt{1}C & \text{Regel: } A \to \mathtt{1}C \\ & \Longrightarrow & \mathtt{10}D & \text{Regel: } C \to \mathtt{0}D \\ & \Longrightarrow & \mathtt{100}D & \text{Regel: } D \to \mathtt{0}D \\ & \vdots & & \\ & \Longrightarrow & & \\ & \Longrightarrow & \mathtt{100\ldots0} & \text{Regel: } D \to \mathtt{0} \end{array}$$

(d) String $\mathtt{00\ldots0100\ldots0}$, *i.e.* der String hat zunächst k `0`en, dann kommt die `1`, dann folgt der Postfix mit weiteren l `0`en, wobei insgesamt mindestens zwei `0`en auftreten müssen. Dazu

$$\begin{array}{rcll} A & \Longrightarrow & \mathtt{0}B & \text{Regel: } A \to \mathtt{0}B \\ & \Longrightarrow & \mathtt{01}D & \text{Regel: } B \to \mathtt{1}D \\ & \Longrightarrow & \mathtt{010} & \text{Regel: } D \to \mathtt{0} \end{array}$$

(c) Die rechtslineare Grammatik, die die Sprache

$$L_3 = \{w \in \Sigma^* \mid w \text{ enthält nicht das Teilwort } \mathtt{110}\}.$$

generiert, ist gegeben durch das 4–Tupel

$$G_3 = (N, T, P, S)$$

mit den Variablen

$$N = \{A, B, C\},$$

dem Terminalalphabet $T = \{\mathtt{0}, \mathtt{1}\}$, der Startvariablen $S = A$ und den Produktionen

$$\begin{array}{lcl} A & \longrightarrow & \mathtt{0}A \mid \mathtt{1}B \mid \mathtt{0} \mid \mathtt{1} \mid \lambda \\ B & \longrightarrow & \mathtt{0}A \mid \mathtt{1}C \mid \mathtt{0} \mid \mathtt{1} \\ C & \longrightarrow & \mathtt{1}C \mid \mathtt{1} \end{array}$$

A.3.2 Übung [3.22]

Geben Sie für die folgenden Sprachen über dem Alphabet $\Sigma = \{a, b\}$ jeweils einen DFA an, der die jeweilige Sprache akzeptiert.

(a) Die Sprache aller Strings, die genau zwei a enthalten.

(b) Die Sprache aller Strings, die wenigstens zwei a enthalten.

(c) Die Sprache aller Strings, die nicht mit ab enden.

(d) Die Sprache aller Strings, die mit aa oder bb beginnen oder enden.

(e) Die Sprache aller Strings, die nicht den Teilstring aa enthalten.

(f) Die Sprache aller Strings, die eine gerade Anzahl von a und eine gerade Anzahl von b enthalten.

(g) Die Sprache aller Strings, die nicht mehr als einmal den Teilstring aa enthalten. (Der String aaa enthält aa zwei Mal)..

(h) Die Sprache aller Strings, bei denen auf jedes a (falls vorhanden) unmittelbar der Teilstring bb folgt.

(i) Die Sprache aller Strings, die bb und aba als Teilstrings enthalten.

(j) Die Sprache aller Strings, die aba und bab als Teilstrings enthalten.

Lösung:

(a) Die Sprache aller Strings über dem Alphabet $\Sigma = \{a, b\}$, die genau zwei a enthalten wird akzeptiert durch den folgenden DFA

$$M_1 = (Q, \Sigma, \delta, q^s, F)$$

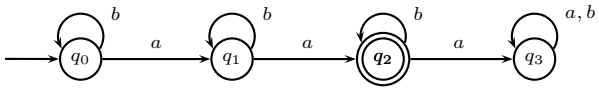
mit

$$Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a, b\}, q^s = q_0, F = \{q_2\}$$

und der Übergangsfunktion δ:

	a	b
q_0	q_1	q_0
q_1	q_2	q_1
q_2	q_3	q_2
q_3	q_3	q_3

Das Zustandsdiagramm dieser Maschine ist



(b) Die Sprache aller Strings, die wenigstens zwei a enthalten, *i.e.* die Strings enthalten zwei oder mehr a. Diese Sprache wird durch den folgenden DFA akzeptiert.

$$M_2 = (Q, \Sigma, \delta, q^s, F)$$

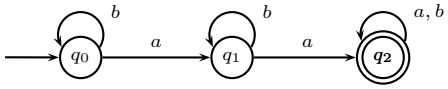
mit

$$Q = \{q_0, q_1, q_2\}, \Sigma = \{a, b\}, q^s = q_0, F = \{q_2\}$$

und der Übergangsfunktion δ:

	a	b
q_0	q_1	q_0
q_1	q_2	q_1
q_2	q_2	q_2

Das Zustandsdiagramm dieser Maschine ist



(c) Die Sprache aller Strings, die nicht mit ab enden, wird von dem folgenden DFA akzeptiert

$$M_3 = (Q, \Sigma, \delta, q^s, F)$$

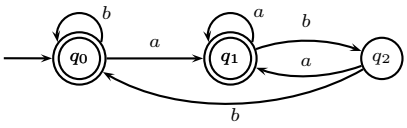
mit

$$Q = \{q_0, q_1, q_2\}, \Sigma = \{a, b\}, q^s = q_0, F = \{q_0, q_1\}$$

und der Übergangsfunktion δ:

	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_0

Das Zustandsdiagramm dieser Maschine ist



Hinweis: Die einfachste Art und Weise, diesen Automaten zu erhalten ist, dass man zunächst den DFA konstruiert, der alle Strings akzeptiert, die mit ab enden. Dann vertauscht man die Nicht–Acceptzustände mit den Acceptzuständen. Dadurch erhält man einen Automaten, der die Komplementärsprache zur ursprünglichen Sprache akzeptiert.

(d) Die Sprache aller Strings, die mit aa oder bb beginnen oder enden, wird von dem folgenden DFA akzeptiert.

$$M_4 = (Q, \Sigma, \delta, q^s, F)$$

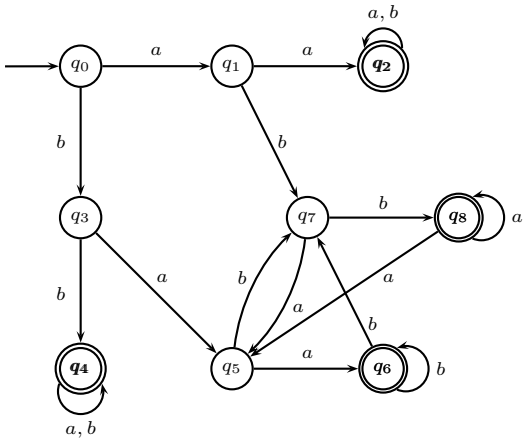
mit

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}, \Sigma = \{a, b\}, q^s = q_0, F = \{q_2, q_4, q_6, q_8\}$$

und der Übergangsfunktion δ:

	a	b
q_0	q_1	q_3
q_1	q_2	q_7
q_2	q_2	q_2
q_3	q_1	q_5
q_4	q_4	q_4
q_5	q_6	q_7
q_6	q_6	q_7
q_7	q_5	q_8
q_8	q_5	q_8

Das Zustandsdiagramm dieser Maschine ist



(e) Die Sprache aller Strings, die nicht den Teilstring aa enthalten, wird von dem folgenden DFA akzeptiert,

$$M_5 = (Q, \Sigma, \delta, q^s, F)$$

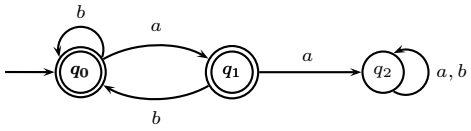
mit

$$Q = \{q_0, q_1, q_2\}, \Sigma = \{a, b\}, q^s = q_0, F = \{q_0, q_1\}$$

und der Übergangsfunktion δ:

	a	b
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2

Das Zustandsdiagramm dieser Maschine ist



(f) Die Sprache aller Strings, die eine gerade Anzahl von a und eine gerade Anzahl von b enthalten, wird von dem folgenden DFA akzeptiert.

$$M_6 = (Q, \Sigma, \delta, q^s, F)$$

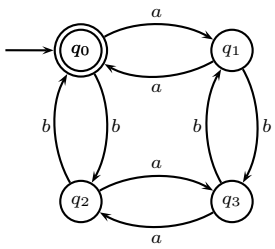
mit

$$Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a, b\}, q^s = q_0, F = \{q_0\}$$

und der Übergangsfunktion δ:

	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_1

Das Zustandsdiagramm dieser Maschine ist



(g) Die Sprache aller Strings, die nicht mehr als einmal den Teilstring aa enthalten[2] wird von dem folgenden DFA akzeptiert. Die Strings enthalten also aa nicht oder genau einmal.

$$M_7 = (Q, \Sigma, \delta, q^s, F)$$

mit

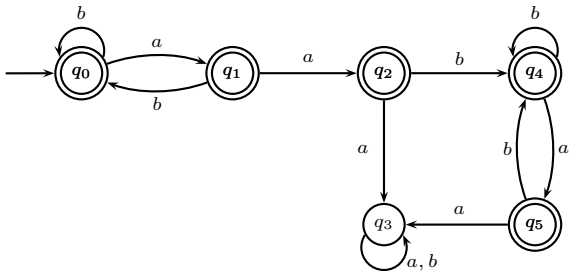
$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}, \Sigma = \{a, b\}, q^s = q_0, F = \{q_0, q_1, q_2, q_4, q_5\}$$

und der Übergangsfunktion δ:

	a	b
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_3	q_4
q_3	q_3	q_3
q_4	q_5	q_4
q_5	q_3	q_4

Das Zustandsdiagramm dieser Maschine ist

[2] Beachte, der String aaa enthält aa zwei Mal



(h) Die Sprache aller Strings, bei denen auf jedes a (falls vorhanden) unmittelbar der Teilstring bb folgt, wird von dem folgenden DFA akzeptiert.

$$M_8 = (Q, \Sigma, \delta, q^s, F)$$

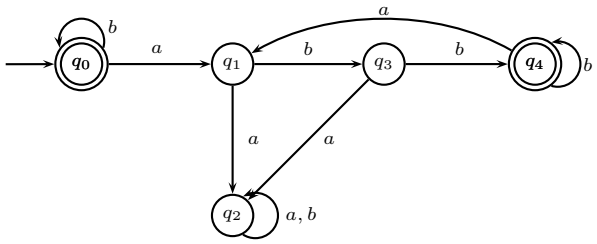
mit

$$Q = \{q_0, q_1, q_2, q_3, q_4\}, \Sigma = \{a, b\}, q^s = q_0, F = \{q_0, q_4\}$$

und der Übergangsfunktion δ:

	a	b
q_0	q_1	q_0
q_1	q_2	q_3
q_2	q_2	q_2
q_3	q_2	q_4
q_4	q_1	q_4

Das Zustandsdiagramm dieser Maschine ist



(i) Die Sprache aller Strings, die bb und aba als Teilstrings enthalten. Diese Sprache wird durch den folgenden DFA akzeptiert. Die Worte sollen bb und aba als Teilstring enthalten. Daher haben die Worte die Form

$$x\ bb\ y\ aba\ z \quad \text{oder } x\ aba\ y\ bb\ z,$$

wobei x, y, z irgendwelche beliebige Strings aus a und b sind.

$$M_9 = (Q, \Sigma, \delta, q^s, F)$$

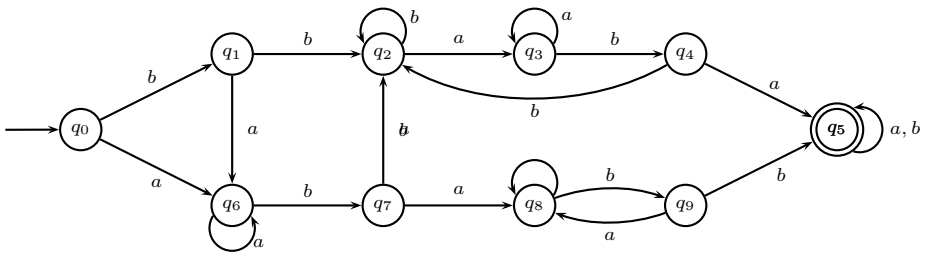
mit zehn Zuständen

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9\}, \Sigma = \{a, b\}, q^s = q_0, F = \{q_5\}$$

und der Übergangsfunktion δ:

	a	b
q_0	q_1	q_6
q_1	q_6	q_2
q_2	q_3	q_2
q_3	q_3	q_4
q_4	q_5	q_2
q_5	q_5	q_5
q_6	q_6	q_7
q_7	q_8	q_2
q_8	q_8	q_9
q_9	q_8	q_5

Das Zustandsdiagramm dieser Maschine sieht folgendermaßen aus:



(j) Die Sprache aller Strings, die *aba* und *bab* als Teilstrings enthalten. Die Worte dieser Sprache enthalten *aba* und *bab* als Teilstring. Daher haben die Worte die Form

$$x\ aba\ y\ bab\ z \quad \text{oder } x\ bab\ y\ aba\ z,$$

wobei x, y, z irgendwelche beliebige Strings aus a und b sind, incl. Leerstring. Es müssen aber auch Strings akzeptiert werden, die den Teilstring *abab* oder *baba* enthalten.

Der `DFA`, der diese Sprache akzeptiert ist:

$$M_{10} = (Q, \Sigma, \delta, q^s, F)$$

mit 16 Zuständen

$$Q = \{q_0, \ldots, q_{15}\},$$

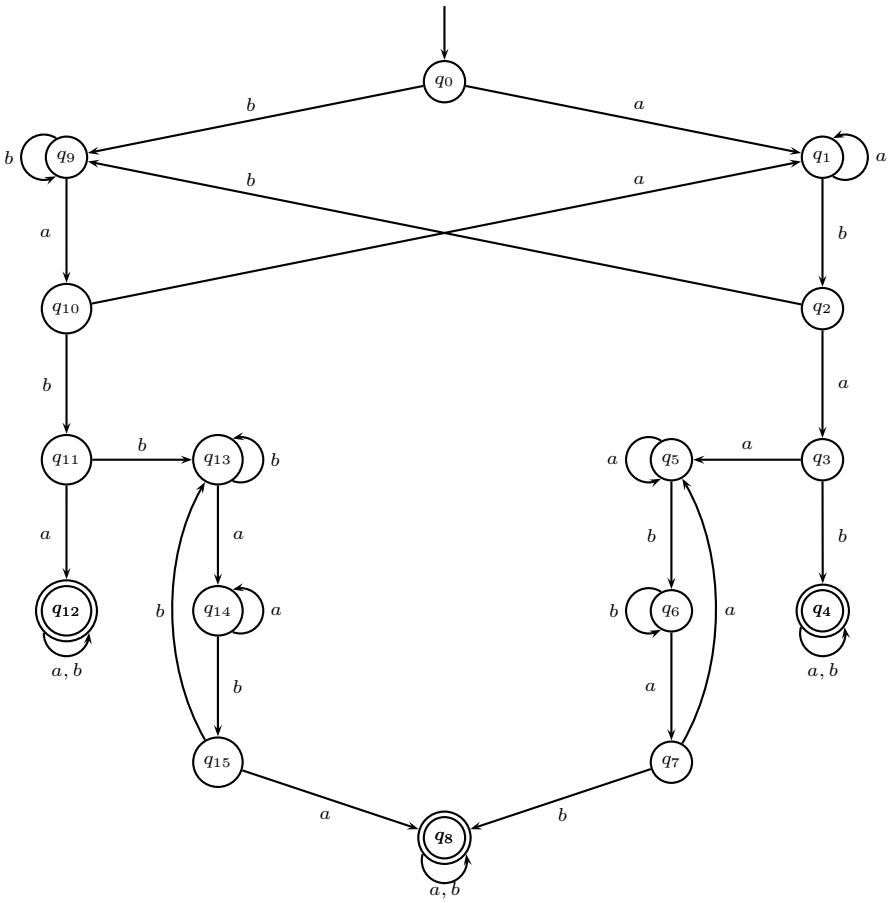
Startzustand ist q_0, Menge der Acceptzustände ist

$$F = \{q_4, q_8, q_{12}\}$$

Die Übergangsfunktion δ ist

	a	b
q_0	q_1	q_9
q_1	q_1	q_2
q_2	q_3	q_9
q_3	q_5	q_4
q_4	q_4	q_4
q_5	q_5	q_6
q_6	q_7	q_6
q_7	q_5	q_8
q_8	q_8	q_8
q_9	q_{10}	q_9
q_{10}	q_1	q_{11}
q_{11}	q_{12}	q_{13}
q_{12}	q_{12}	q_{12}
q_{13}	q_{14}	q_{13}
q_{14}	q_{14}	q_{15}
q_{15}	q_8	q_{13}

Das Zustandsdiagramm ist:


q_0
q_9
q_1
q_{10}
q_2
q_{11}
q_{13}
q_5
q_3
q_{12}
q_{14}
q_6
q_4
q_{15}
q_7
q_8
a
b
a, b

A.3.3 Übung [3.23]

Geben Sie einen nichtdeterministischen endlichen Automaten über dem unären Alphabet $\Sigma = \{a\}$ an, der die Sprache

$$L = \{w \in \Sigma^* \mid w = a^n, n \text{ ist durch 3 oder 5 teilbar}\}.$$

erkennt.

Lösung:

Definiere

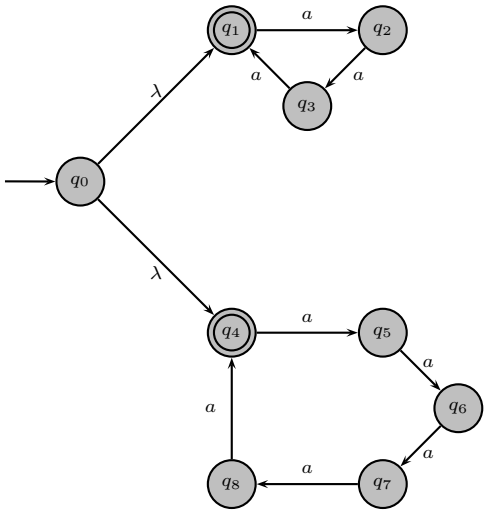
$$N = (Q, \Sigma, \delta, q^s, F)$$

wie folgt:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$
- $\Sigma = \{a\}$
- $q^s = q_0$
- $F = \{q_1, q_4\}$
- Die Übergangsrelation δ ist:

	a	λ
q_0	$\emptyset$	$\{q_1, q_4\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$
q_3	$\{q_1\}$	$\emptyset$
q_4	$\{q_5\}$	$\emptyset$
q_5	$\{q_6\}$	$\emptyset$
q_6	$\{q_7\}$	$\emptyset$
q_7	$\{q_8\}$	$\emptyset$
q_8	$\{q_4\}$	$\emptyset$

Das Zustandsdiagramm dieses `NFA` ist:



Wir können einen äquivalenten `NFA` angeben, der keine λ–Übergänge hat.

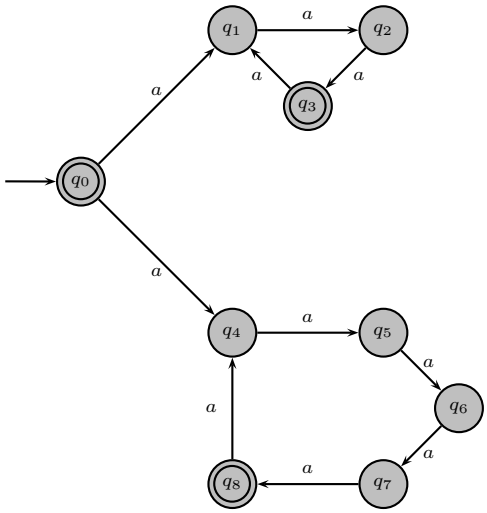
$$N' = (Q, \Sigma, \delta, q^s, F)$$

wie folgt:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$
- $\Sigma = \{a\}$
- $q^s = q_0$
- $F = \{q_0, q_3, q_8\}$
- Die Übergangsrelation δ ist:

	a
q_0	$\{q_1, q_4\}$
q_1	$\{q_2\}$
q_2	$\{q_3\}$
q_3	$\{q_1\}$
q_4	$\{q_5\}$
q_5	$\{q_6\}$
q_6	$\{q_7\}$
q_7	$\{q_8\}$
q_8	$\{q_4\}$

Das Zustandsdiagramm dieses Automaten ist:


q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
q_8
a

A.3.4 Übung [3.24]

Geben Sie zu jeder der folgenden Sprachen einen NFA an (siehe Übung [3.21]), der die Sprache erkennt.

(a)

$$L_1 = \{w \in \Sigma^* \mid w \text{ beginnt mit einer ungeraden Anzahl } \texttt{0}\text{en}, \text{ gefolgt von einer geraden Anzahl } \texttt{1}\text{en}\}.$$

(b) $L_2 = \{w \in \Sigma^* \mid w \text{ enthält mindestens zwei } \texttt{0}\text{en und höchstens eine } \texttt{1}\}$

(c) $L_3 = \{w \in \Sigma^* \mid w \text{ enthält nicht das Teilwort } \texttt{110}\}$.

Leiten Sie aus den Automaten mit dem in dem Beweis von Satz [1] verwendeten Verfahren eine reguläre Grammatik ab. Vergleichen Sie diese mit der Übung [3.21].

Lösung:

(a) Der folgende NFA akzeptiert die Sprache

$$L_1 = \{w \in \{\texttt{0},\texttt{1}\}^* \mid \texttt{w} = \texttt{0}^{\texttt{n}}\texttt{1}^{\texttt{m}}, \texttt{n} = \texttt{2k}+\texttt{1}, \texttt{m} = \texttt{2l}, \texttt{k}, \texttt{l} = \texttt{0},\texttt{1},\texttt{2},\ldots\}.$$

Die Bitstrings dieser Sprache haben die Form

$$L_1 = \{\texttt{0}, \texttt{011}, \texttt{01111}, \texttt{0111111}, \texttt{000}, \texttt{00011}, \texttt{0001111}, \ldots\}$$

Der NFA ist gegeben durch das Quintupel

$$N = (Q, \Sigma, \delta, q^s, F)$$

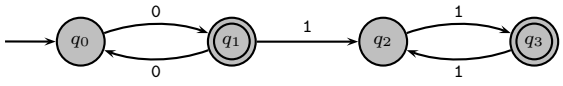
mit den vier Zuständen

$$Q = \{q_0, q_1, q_2, q_3\}, \qquad \Sigma = \{\texttt{0},\texttt{1}\},$$

dem Startzustand $q^s = q_0$, der Menge der Acceptzustände $F = \{q_1, q_3\}$. Die Übergangsrelation δ ist

	0	1
q_0	$\{q_1\}$	$\emptyset$
q_1	$\{q_0\}$	$\{q_2\}$
q_2	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_2\}$

Das Zustandsdiagramm dieses Automaten ist:



(b) Die Sprache

$$L_2 = \{w \in \Sigma^* \mid w \text{ enthält mindestens zwei } \texttt{0}\text{en und höchstens eine } \texttt{1}\}$$

enthält Worte der Form

$$\texttt{00}, \texttt{000}, \texttt{001}, \texttt{010}, \texttt{100}, \texttt{0000}, \texttt{0001}, \texttt{0010}, \texttt{0100}, \texttt{1000}, \ldots$$

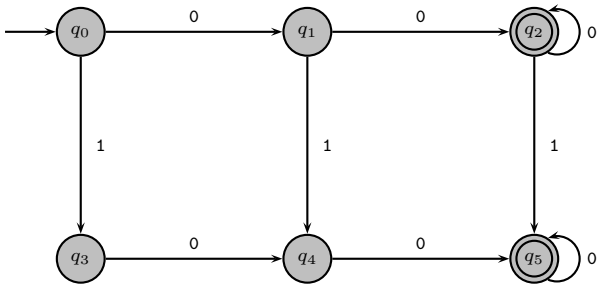
Der `NFA` hat folgenden Aufbau:

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\},$$

Startzustand ist q_0, Alphabet ist $\Sigma = \{\texttt{0}, \texttt{1}\}$, Acceptzustände ist die Menge $F = \{q_2, q_5\}$ und die Übergangsrelation ist

	`0`	`1`
q_0	$\{q_1\}$	$\{q_3\}$
q_1	$\{q_2\}$	$\{q_4\}$
q_2	$\{q_2\}$	$\{q_5\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_5\}$	$\emptyset$
q_5	$\{q_5\}$	$\emptyset$

Das Zustandsdiagramm dieses Automaten ist:



(c) Die Sprache

$$L_3 = \{w \in \Sigma^* \mid w \text{ enthält nicht das Teilwort } \texttt{110}\}$$

wird durch den folgenden Automaten akzeptiert. Dies ist ein `DFA`, da aber jeder `DFA` auch ein `NFA` ist, ist dies legitim.

$$M = (Q, \Sigma, \delta, q^s, F),$$

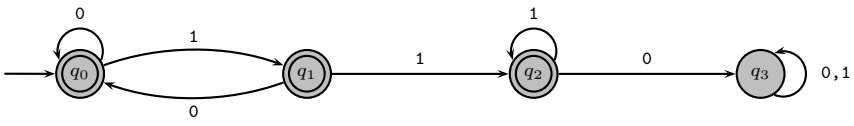
mit vier Zuständen

$$Q = \{q_0, q_1, q_2, q_3\},$$

Inputalphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$, Startzustand $q^s = q_0$, Acceptzustände $F = \{q_0, q_1, q_2\}$ und der Übergangsfunktion δ:

	0	1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_3	q_2
q_3	q_3	q_3

Das Zustandsdiagramm dieses Automaten ist:



Wir konstruieren nun rechtslineare Grammatiken aus den jeweiligen Automaten.

(a) Der `NFA`, der die Sprache L_1 akzeptiert, ist gegeben durch

$$N = (Q, \Sigma, \delta, q^s, F)$$

mit

$$Q = \{q_0, q_1, q_2, q_3\}, \qquad \Sigma = \{\texttt{0}, \texttt{1}\},$$

dem Startzustand $q^s = q_0$, $F = \{q_1, q_3\}$, und δ ist

	0	1
q_0	$\{q_1\}$	$\emptyset$
q_1	$\{q_0\}$	$\{q_2\}$
q_2	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_2\}$

Wir konstruieren eine rechstlineare Grammatik

$$G_1 = (N, T, P, S).$$

Wir ordnen jedem Zustand $q_i \in Q$ eine Variable zu:

$$\begin{array}{lcl} q_0 & \longleftrightarrow & S \\ q_1 & \longleftrightarrow & A \\ q_2 & \longleftrightarrow & B \\ q_3 & \longleftrightarrow & C. \end{array}$$

Das Terminalalphabet ist: $T = \{\mathtt{0}, \mathtt{1}\}$, Startvariable ist S und aus der Übergangsrelation erhalten wir die folgenden Produktionen:

$$\begin{array}{rcl} \delta(q_0, \mathtt{0}) = \{q_1\} & \Longrightarrow & S \longrightarrow \mathtt{0}A \\ \delta(q_0, \mathtt{1}) = \emptyset & \Longrightarrow & \text{keine Regel} \\ \delta(q_1, \mathtt{0}) = \{q_0\} & \Longrightarrow & A \longrightarrow \mathtt{0}S \\ \delta(q_1, \mathtt{1}) = \{q_2\} & \Longrightarrow & A \longrightarrow \mathtt{1}B \\ \delta(q_2, \mathtt{0}) = \emptyset & \Longrightarrow & \text{keine Regel} \\ \delta(q_2, \mathtt{1}) = \{q_3\} & \Longrightarrow & B \longrightarrow \mathtt{1}C \\ \delta(q_3, \mathtt{0}) = \emptyset & \Longrightarrow & \text{keine Regel} \\ \delta(q_3, \mathtt{1}) = \{q_2\} & \Longrightarrow & C \longrightarrow \mathtt{1}B \end{array}$$

Da q_1 unds q_3 Acceptszustände sind, werden die Regeln $A \longrightarrow \lambda$ und $C \longrightarrow \lambda$ hinzugefügt. damit sind die Produktionen P der Grammatik:

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}A, \\ A & \longrightarrow & \mathtt{0}S \mid \mathtt{1}B \mid \lambda, \\ B & \longrightarrow & \mathtt{1}C, \\ C & \longrightarrow & \mathtt{1}B \mid \lambda. \end{array}$$

Wir können die λ–Regeln eliminieren:

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}A \mid \mathtt{0}, \\ A & \longrightarrow & \mathtt{0}S \mid \mathtt{1}B, \\ B & \longrightarrow & \mathtt{1}C \mid \mathtt{1}, \\ C & \longrightarrow & \mathtt{1}B. \end{array}$$

(b) Der `NFA`, der die Sprache L_2 akzeptiert ist:

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\},$$

Startzustand ist q_0, Alphabet ist $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, Acceptzustände ist die Menge $F = \{q_2, q_5\}$ und die Übergangsrelation ist

	`0`	`1`
q_0	$\{q_1\}$	$\{q_3\}$
q_1	$\{q_2\}$	$\{q_5\}$
q_2	$\{q_2\}$	$\{q_5\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_5\}$	$\emptyset$
q_5	$\{q_5\}$	$\emptyset$

Hieraus erhalten wir die folgende rechtslineare Grammatik $G_2 = (N, T, P, S)$: Jedem Zustand $q_i \in Q, i = 0, \ldots, 5$ ordnen wir eine Variable zu:

$$N = \{S, A, B, C, D, E\}.$$

Das Terminalalphabet ist $T = \{\mathtt{0}, \mathtt{1}\}$, die Produktionen folgen aus der Übergangsrelation des Automaten; da q_2 und q_5 Acceptzustände sind, benötigen wir zwei entsprechende λ–Übergänge:

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}A \mid \mathtt{1}C, \\ A & \longrightarrow & \mathtt{0}B \mid \mathtt{1}E, \\ B & \longrightarrow & \mathtt{0}B \mid \mathtt{1}E \mid \lambda, \\ C & \longrightarrow & \mathtt{0}D, \\ D & \longrightarrow & \mathtt{0}E, \\ E & \longrightarrow & \mathtt{0}E \mid \lambda. \end{array}$$

Hier lassen sich auch wieder die λ–Übergänge eliminieren, ohne dass sich die Sprache ändert, die die Grammatik generiert:

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}A \mid \mathtt{1}C, \\ A & \longrightarrow & \mathtt{0}B \mid \mathtt{1}E \mid \mathtt{0} \mid \mathtt{1}, \\ B & \longrightarrow & \mathtt{0}B \mid \mathtt{1}E \mid \mathtt{1}, \\ C & \longrightarrow & \mathtt{0}D, \\ D & \longrightarrow & \mathtt{0}E \mid \mathtt{0}, \\ E & \longrightarrow & \mathtt{0}E. \end{array}$$

(c) Die Sprache L_3 wird durch den folgenden `DFA` akzeptiert

$$M = (Q, \Sigma, \delta, q^s, F),$$

mit $Q = \{q_0, q_1, q_2, q_3\}$, Inputalphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, Startzustand $q^s = q_0$, Acceptzustände $F = \{q_0, q_1, q_2\}$ und der Übergangsfunktion δ:

	`0`	`1`
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_3	q_2
q_3	q_3	q_3

Gemäß dem Standardverfahren konstruieren wir eine rechtslineare Grammatik G_3. Jedem Zustand wird eine Variable zugeordnet, wir nennen diese

$$N = \{S, A, B, C\},$$

die Startvariable ist S, das Terminalalphabet ist $\mathtt{0}, \mathtt{1}$ und die Produktionen ergeben sich aus der obigen Übergangsfunktion δ:

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}S \mid \mathtt{1}A \\ A & \longrightarrow & \mathtt{0}S \mid \mathtt{1}B \\ B & \longrightarrow & \mathtt{0}C \mid \mathtt{1}B \\ C & \longrightarrow & \mathtt{0}C \mid \mathtt{1}C. \end{array}$$

Die Zustände q_0, q_1, q_2 sind Acceptzustände, also fügen wir noch die Produktionen

$$S \longrightarrow \lambda, A \longrightarrow \lambda \text{ und } B \longrightarrow \lambda$$

hinzu. Diese können wir wieder eliminieren, mit Ausnahme des $S \longrightarrow \lambda$–Übergangs, und erhalten:

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}S \mid \mathtt{1}A \mid \lambda \mid \mathtt{1} \\ A & \longrightarrow & \mathtt{0}S \mid \mathtt{1}B \mid \mathtt{0} \mid \mathtt{1} \\ B & \longrightarrow & \mathtt{0}C \mid \mathtt{1}B \mid \mathtt{1} \\ C & \longrightarrow & \mathtt{0}C \mid \mathtt{1}C. \end{array}$$

Wir können die Grammatik noch weiter vereinfachen, denn die Variable C ist überflüssig. Damit

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}S \mid \mathtt{1}A \mid \lambda \mid \mathtt{1} \\ A & \longrightarrow & \mathtt{0}S \mid \mathtt{1}B \mid \mathtt{0} \mid \mathtt{1} \\ B & \longrightarrow & \mathtt{1}B \mid \mathtt{1}. \end{array}$$

A.3.5 Übung [3.25]

Die Sprache

$$L = \{w \in \{a,b\}^* \mid w = (ab)^n a (ba)^n, n \geq 0\}$$

ist regulär.

(a) Geben Sie die Wörter für $n = 0, 1, 2, 3$ an.

(b) Geben Sie einen nichtdeterministischen oder deterministischen endlichen Automaten an, der diese Sprache akzeptiert.

(c) Beschreiben Sie L mit Hilfe eines regulären Ausdrucks.

Lösung:

Es gilt:

$$\begin{aligned} n = 0 &: w = a \\ n = 1 &: w = ababa \\ n = 2 &: w = abababababa \\ n = 3 &: w = ababababababababa. \end{aligned}$$

Ein `DFA`, der diese Sprache akzeptiert, ist

$$M = (Q, \{a,b\}, \delta, q^s, F)$$

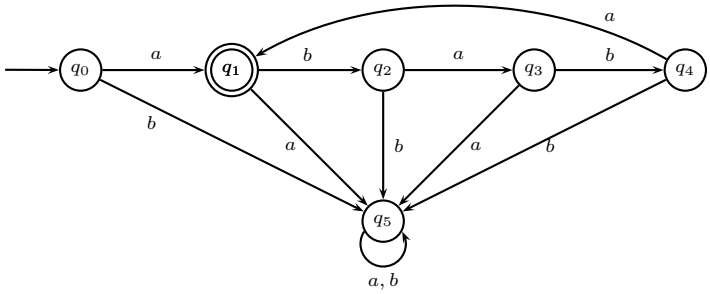
mit der Menge der Zustände

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

dem Startzustand $q^s = q_0$, der Menge der Acceptzustände $F = \{q_1\}$ und der Übergangsfunktion δ:

	a	b
q_0	q_1	q_5
q_1	q_5	q_2
q_2	q_3	q_5
q_3	q_5	q_4
q_4	q_1	q_5
q_5	q_5	q_5

Das Zustandsdiagramm ist:



Ein regulärer Ausdruck, der diese Sprache generiert ist

$$R = (\mathtt{abab})^*\mathtt{a}.$$

A.3.6 Übung [3.26]

Betrachten Sie den folgenden NFA, der alle Bitstrings akzeptiert, die mit 00 enden.

$$N = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$

mit der Übergangsrelation δ:

	0	1
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\emptyset$

(a) Konstruieren Sie einen äquivalenten DFA mit Hilfe des Potenzmengenverfahrens.

(b) Konstruieren Sie einen äquivalenten DFA über den Zustandsbaum.

Lösung:

(a) Wir konstruieren einen DFA, der die gleiche Sprache akzeptiert wie der NFA N. Wir setzen

$$M = (Q, \Sigma, \delta, q^s, F),$$

mit der Zustandsmenge

$$\begin{aligned} Q &= \wp(\{q_0, q_1, q_2\}) \\ &= \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}. \end{aligned}$$

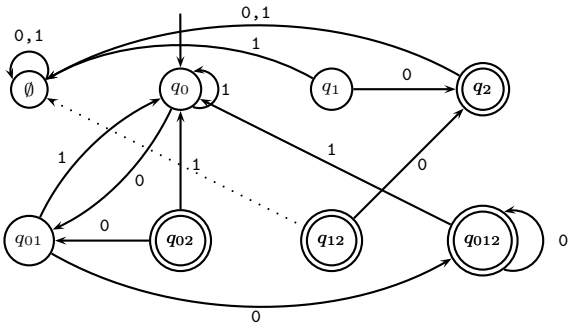
Startzustand ist $q^s = \{q_0\}$, die Menge der Acceptzustände ist:

$$F = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}.$$

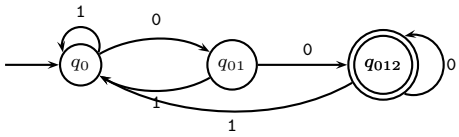
Die Übergangsfunktion δ wird:

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\{q_2\}$	$\emptyset$
$\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1, q_2\}$	$\{q_2\}$	$\emptyset$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

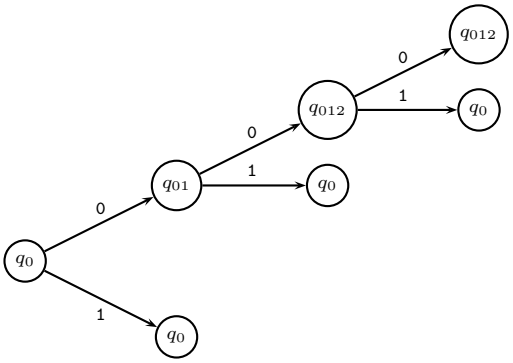
Damit erhält man das Zustandsdiagramm des DFA zu: (Wie bezeichnen der Übersichtlichkeit halber die Zustände mit Indices, *e.g.* $\{q_0, q_1, q_2\}$ bezeichnen wir mit q_{012}).



Es ist leicht zu sehen, dass dieser DFA vereinfacht werden kann, ohne die akzeptierte Sprache zu ändern. Die Zustände $\emptyset, q_1, q_2, q_{02}, q_{12}$ können nicht vom Startzustand aus erreicht werden. Das führt auf den folgenden DFA (Minimalautomat):



(b) Um über das Verfahren des Zustandsbaums den DFA zu konstruieren, betrachten wir die Zustandänderungen des NFA, beginnend mit dem Startzustand q_0.



Der Zustandsbaum kommt folgendermaßen zustande:

(a) Der NFA startet im Startzustand q_0. Liest er eine 0, geht er in die beiden Zustände q_0 und q_1 über. Das entspricht der Zuordnung $q_0 \longrightarrow q_{01}$. Liest er eine 1, geht er in den Zustand q_0 über. Das ist die Zuordnung $q_0 \longrightarrow q_0$.

(b) Die zweite Ebene: Ist der Automat in q_{01}, *i.e.* entweder in q_0 oder q_1, dann geht er beim Lesen einer `0` in q_0, q_1 über oder in q_2, also in q_{012}.

(c) In der dritten Ebene ist der Zustand q_{012} zu betrachten. Wenn eine `0` verarbeitet wird, geht der Automat wieder in den Zustand q_{012} über, bei einer `1` in den Zustand q_0.

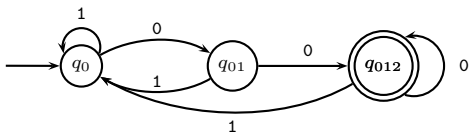
Mit dem Zustandsbaum erhalten wir sukzessiv alle Zustandsmengen, die vom Startzustand aus erreichbar sind. Aus dem Zustandsbaum lässt sich direkt der `DFA` konstruieren:

$$q_0 \longrightarrow \begin{cases} \{q_0, q_1\} & \text{falls eine } \texttt{0} \text{ gelesen wird,} \\ \{q_0\} & \text{falls eine } \texttt{1} \text{ gelesen wird.} \end{cases}$$

$$q_{01} \longrightarrow \begin{cases} \{q_0, q_1, q_2\} & \text{falls eine } \texttt{0} \text{ gelesen wird,} \\ \{q_0\} & \text{falls eine } \texttt{1} \text{ gelesen wird.} \end{cases}$$

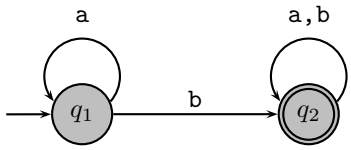
$$q_{012} \longrightarrow \begin{cases} \{q_0, q_1, q_2\} & \text{falls eine } \texttt{0} \text{ gelesen wird,} \\ \{q_0\} & \text{falls eine } \texttt{1} \text{ gelesen wird.} \end{cases}$$

Damit erhalten wir das folgende Zustandsdiagramm des `DFA`:



A.3.7 Übung [3.27]

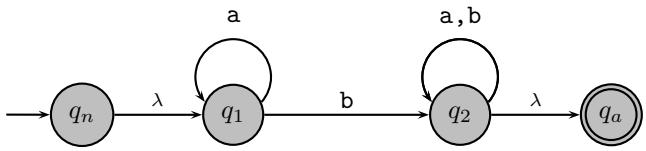
Gegeben ist der folgende DFA.



Konstruieren Sie aus diesem Automaten einen regulären Ausdruck. Verwenden Sie dazu das Verfahren, das im Beweis des Satzes [3.17] angegeben wurde.

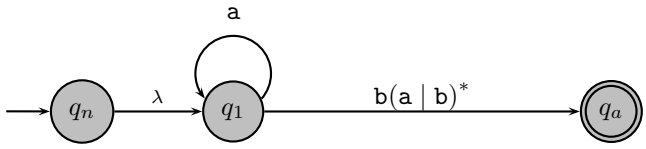
Lösung:

In einem ersten Schritt wird ein neuer Startzustand und ein neuer Acceptzustand hinzugefügt:

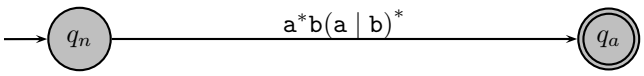


Der Acceptzustand des ursprünglichen Automaten wird zu einem normalen Zustand.

In einem nächsten Schritt eliminieren wir den Zustand q_2.

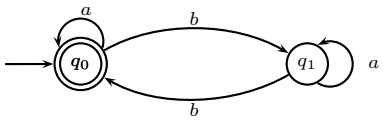


Schließlich wird der Zustand q_1 eliminiert.


q_n
$\mathtt{a}^*\mathtt{b}(\mathtt{a} \mid \mathtt{b})^*$
q_a

A.3.8 Übung [3.28]

Gegeben ist der folgende DFA, der alle Wörter über dem Alphabet $\Sigma = \{a, b\}$ akzeptiert, die eine gerade Anzahl von bs enthalten.

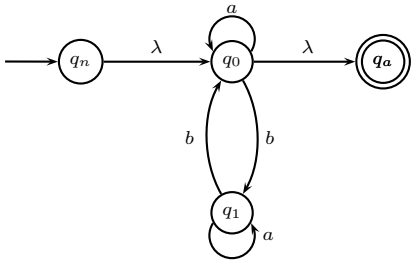


Konstruieren Sie aus diesem Automaten einen regulären Ausdruck. Verwenden Sie dazu das Verfahren, das im Beweis des Satzes [3.17] angegeben wurde.

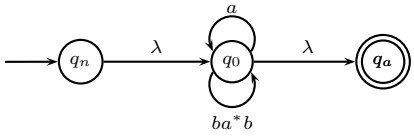
Lösung:

Wir verfahren wie folgt, um einen regulären Ausdruck zu konstruieren, der die gleiche Sprache generiert.

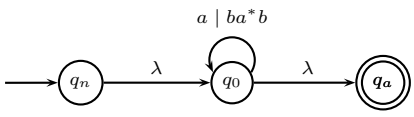
Hinzufügen eines neuen Startzustandes q_n sowie eines Acceptzustandes q_a, die alten Acceptzustände werden normale Zustände.



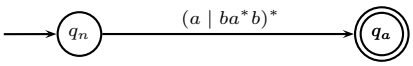
Elimination des Zustands q_1.



Elimination der Doppelkante des Zustands q_0. Die beiden Loops, die in q_0 zurücklaufen, ersetzen wir durch einen Loop, die Ausdrücke, die die Loops beschriften, sind dann Alternativen.



Elimination des Zustandes q_0. Die Loop–Beschriftung wird mit der KLEENE Stern oeration erweitert,

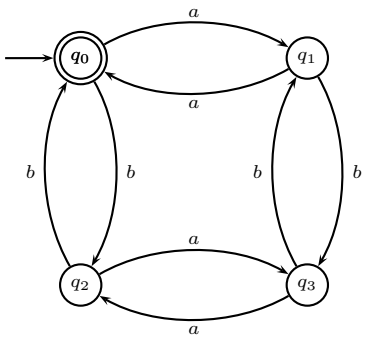


Damit lautet der reguläre Ausdruck:

$$R = (a \mid ba^*b)^*.$$

A.3.9 Übung [3.29]

Gegeben ist der folgende DFA, der alle Wörter über dem Alphabet $\Sigma = \{a, b\}$ akzeptiert, die eine gerade Anzahl von as und bs enthalten.

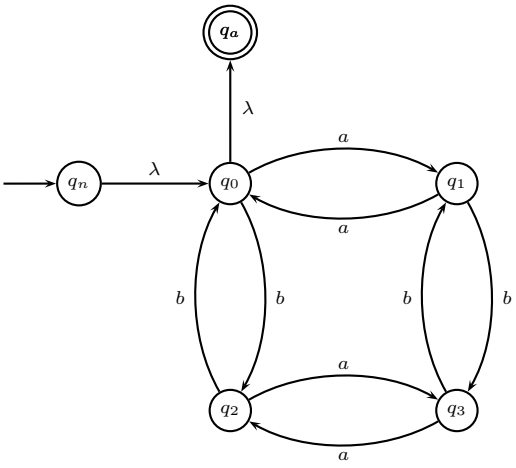


Konstruieren Sie aus diesem Automaten einen regulären Ausdruck. Verwenden Sie dazu das Verfahren, das im Beweis des Satzes [3.17] angegeben wurde.

Lösung:

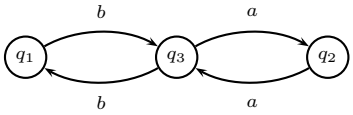
Wir verfahren wie folgt, um einen regulären Ausdruck zu konstruieren, der die gleiche Sprache generiert.

Wir fügen eines neuen Startzustand sowie einen Acceptzustand hinzu, die alten Acceptzustände werden normale Zustände. Damit ergibt sich das folgende modifizierte Zustandsdiagramm



Wir eliminieren den Zustand q_3.

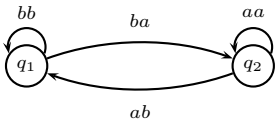
Der Übersichtlichkeit halber, betrachten wir den Teilgraphen, der aus den Zuständen q_1, q_3 und q_2 besteht.



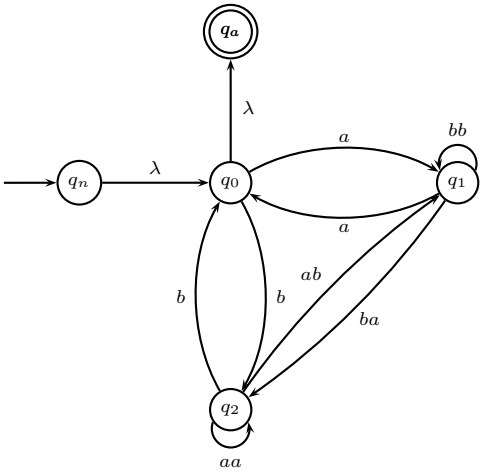
Um den Zustand q_3 zu eliminieren, müssen wir alle möglichen Kantenzüge betrachten:

- $q_1 \xrightarrow{b} q_3 \xrightarrow{b} q_1$: liefert einen Loop, der Zustand q_1 geht in sich über mit der Beschriftung bb.
- $q_1 \xrightarrow{b} q_3 \xrightarrow{a} q_2$: liefert einen Übergang vom Zustand q_1 in den Zustand q_2 mit der Beschriftung ba.
- $q_2 \xrightarrow{a} q_3 \xrightarrow{a} q_2$: liefert einen Loop, der Zustand q_2 geht in sich über mit der Beschriftung aa.
- $q_2 \xrightarrow{a} q_3 \xrightarrow{b} q_1$: liefert einen Übergang vom Zustand q_2 in den Zustand q_1 mit der Beschriftung ab.

Die Elimination des Zustands q_3 liefert daher den folgenden Teilgraphen:

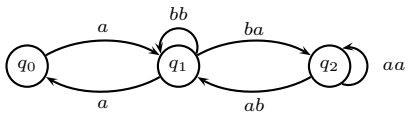


Damit erhalten wir:



Wir eliminieren den Zustand q_1. i

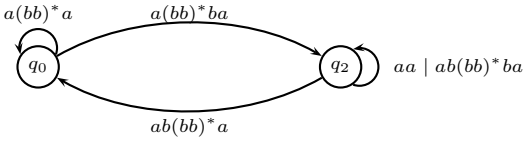
Um den Zustand q_1 zu eliminieren, betrachten wir wieder den Teilgraphen, bestehend aus den drei Zuständen q_0, q_1 und q_2.



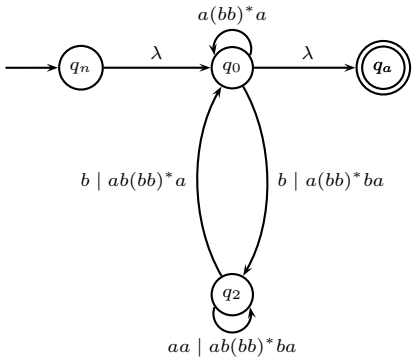
In diesem Teilgraphen müssen wir wieder alle Pfade finden:

- $q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_0$: liefert einen Loop, der Zustand q_0 geht in sich über mit der Beschriftung $a(bb)^*a$.
- $q_0 \xrightarrow{a} q_1 \xrightarrow{ba} q_2$: liefert einen Übergang vom Zustand q_0 in q_2 mit der Beschriftung $a(bb)^*ba$.
- $q_2 \xrightarrow{ab} q_1 \xrightarrow{ba} q_2$: liefert einen Loop, der Zustand q_2 geht in sich über mit der Beschriftung $ab(bb)^*ba$. Dieser Loop ist parallel zu dem vorhandenen aa Loop.
- $q_2 \xrightarrow{ab} q_1 \xrightarrow{a} q_0$: liefert einen Übergang vom Zustand q_2 in q_0 mit der Beschriftung $ab(bb)^*a$.

Damit erhalten wir durch die Elimination von q_1 den Teilgraphen:

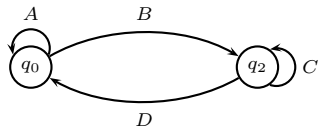


Damit erhalten wir:



Elimination des Zustands q_2

Um den Zustand q_2 zu eliminieren, betrachten wir wieder den Teilgraphen, bestehend aus den beiden Zuständen q_0 und q_2.



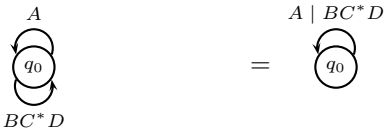
Der Übersichtlichkeit setzen wir für die Kantenbeschriftungen:

$$\begin{aligned} A &= a(bb)^*a \\ B &= b \mid a(bb)^*ba \\ C &= aa \mid ab(bb)^*ba \\ D &= b \mid ab(bb)^*a. \end{aligned}$$

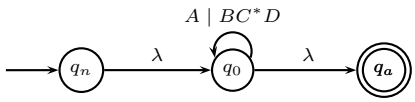
In diesem Teilgraphen müssen wir wieder alle Pfade finden:

- $q_0 \xrightarrow{B} q_2 \xrightarrow{C} q_2 \xrightarrow{D} q_0$: liefert einen Loop, der Zustand q_0 geht in sich über mit der Beschriftung BC^*D.
- $q_0 \xrightarrow{A} q_0$: liefert einen weiteren Loop, der Zustand q_0 geht in sich über mit der Beschriftung A. Dieser Loop ist parallel zu dem obigen Loop.

Damit erhalten wir durch die Elimination von q_2 den Teilgraphen:



Damit erhalten wir:



Schließlich wird der Zustand q_0 eliminiert, Wir erhalten dann:

Wir erhalten damit den regulären Ausdruck

$$\begin{aligned} R &= (A \mid BC^*D)^* \\ &= (a(bb)^*a \mid b \mid a(bb)^*ba(aa \mid ab(bb)^*ba)^* \mid b \mid ab(bb)^*a)^* \,. \end{aligned}$$

Diesen regulären Ausdruck können wir noch etwas übersichtlicher gestalten, denn die Teilausdrücke A und C sind gleich, sowie B und D. Damit können wir schreiben

$$\begin{aligned} R &= (A \mid BA^*B)^* \\ &= (a(bb)^*a \mid b \mid (a(bb)^*ba)(a(bb)^*ba)^*(b \mid a(bb)^*ba))^* \,. \end{aligned}$$

Hieraus ist einfacher zu sehen, dass dieser reguläre Ausdruck die Strings

$$aa, bb, aabb, abab, baba, bbaa, \text{ usw.}$$

erzeugt.

A.3.10 Übung [3.30]

Für einen Binärstring $w \in \{\mathtt{0},\mathtt{1}\}^+$ bezeichne w_2 den Zahlenwert des Binärstrings als Dualzahl interpretiert.

(a) Entwerfen Sie einen deterministischen endlichen Automaten M_1, der alle Binärstrings akzeptiert, die als Dualzahl interpretiert durch 2 teilbar sind. Der Automat soll also die folgende Sprache akzeptieren:

$$L_1 = \{w \in \{\mathtt{0},\mathtt{1}\}^+ \mid w_2 \bmod 2 \equiv 0\}.$$

(b) Entwerfen Sie einen deterministischen endlichen Automaten M_2, der alle Binärstrings akzeptiert, die als Dualzahl interpretiert durch 4 teilbar sind. Der Automat soll also die folgende Sprache akzeptieren:

$$L_2 = \{w \in \{\mathtt{0},\mathtt{1}\}^+ \mid w_2 \bmod 4 \equiv 0\}.$$

Lösung:

(a) Die Sprache

$$L_1 = \{w \in \{\mathtt{0},\mathtt{1}\}^+ \mid w_2 \bmod 2 \equiv 0\}$$

besteht aus allen Binärstrings, die mit $\mathtt{0}$ enden. Wenn diese Binärstrings als Dualzahl interpretiert werden, sind dies die geraden Zahlen, die ohne Rest durch 2 teilbar sind. Da die Dezimalzahl $z = 0_{10}$ ebenfalls ohne Rest durch 2 teilbar ist, enthält L_1 auch den Bitstring $\mathtt{0000}\cdots$ Man beachte, dass das leere Wort ausgenommen ist. Der `DFA`, der diese Sprache akzeptiert ist formal:

$$M_1 = (Q, \{\mathtt{0},\mathtt{1}\}, \delta, q^s, F),$$

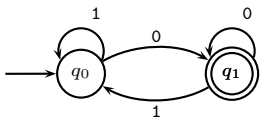
mit

$$Q = \{q_0, q_1\}, q^s = q_0, \quad F = \{q_1\}$$

und δ:

$$\begin{array}{lcl} (q_0,\mathtt{0}) & \longrightarrow & q_1 \\ (q_0,\mathtt{1}) & \longrightarrow & q_0 \\ (q_1,\mathtt{0}) & \longrightarrow & q_1 \\ (q_1,\mathtt{1}) & \longrightarrow & q_0 \end{array}$$

Das Zustandsdiagramm dieses Automaten ist:



(b) Die Sprache

$$L_2 = \{w \in \{\mathtt{0},\mathtt{1}\}^+ \mid w_2 \bmod 4 \equiv 0\}$$

beschreibt alle Bitstrings, die Zahlen codieren, die ohne Rest durch 4 teilbar sind. Dies sind alle Bitstrings, die mit `00` enden. Da die Dezimalzahl $z = 0_{10}$ ebenfalls ohne Rest durch 4 teilbar ist, enthält L_2 auch den Bitstring $\mathtt{0000}\cdots$. Der `DFA`, der diese Sprache erkennt ist

$$M = (Q, \mathtt{0}, \mathtt{1}, \delta, q^s, F)$$

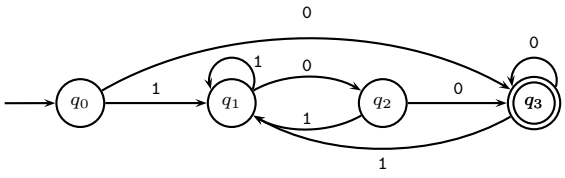
mit der Menge der Zustände

$$Q = \{q_0, q_1, q_2, q_3\}$$

sowie $q^2 = q_0, F = \{q_3\}$ und den Übergängen

	0	1
q_0	q_3	q_1
q_1	q_2	q_1
q_2	q_3	q_1
q_3	q_3	q_1

Das Zustandsdiagramm ist:



A.3.11 Übung [3.31]

Geben Sie jeweils einen nichtdeterministischen endlichen Automaten (NFA) mit der angegeben Anzahl von Zuständen an, der die jeweiligen Sprachen erkennt.

1. Die Sprache $L_1 = \{w \mid w \text{ endet mit } \texttt{00}\}$ mit drei Zuständen.
2. Die Sprache

   $$L_2 = \{w \mid w \text{ enthält den Substring } \texttt{0101}\}$$

   mit fünf Zuständen.
3. Die Sprache

   $$L_3 = \{w \mid w \text{ enthält eine gerade Anzahl von } \texttt{0}\text{en oder genau zwei } \texttt{1}\text{en}\}$$

   mit sechs Zuständen.
4. Die Sprache $L_4 = \{0\}$ mit zwei Zuständen.
5. Die Sprache $L_5 = \texttt{0}^*\texttt{1}^*\texttt{0}^*\texttt{1}^*$ mit vier Zuständen.
6. Die Sprache $L_6 = \{\lambda\}$ mit einem Zustand.
7. Die Sprache $L_7 = \texttt{0}^*$ mit einem Zustand.

Lösung:

Zu 1 Der zu konstruierende nichtdeterministische endliche Automat NA_1 soll mit drei Zuständen sämtliche Strings akzeptieren, die mit dem Substring 00 enden.

Liest der Automat einen zu verarbeitenden String ein, startet er im Zustand q_1 und geht unter 0,1 einfach in sich über. Gleichzeitig hat der q_1 Zustand einen 0-Übergang in einen Zustand q_2, denn die erste gelesene 0 könnte bereits die vorletzte 0 sein.

Unter 0 geht der Zustand q_2 in einen Zustand q_3 über, dieser ist Acceptzustand, da beim Erreichen dieses Zustands zwei 0en gelesen wurden. Wird nun kein weiteres Zeichen gelesen, ist der nichtdeterministische Automat im Zustand q_3 und akzeptiert den String. Wird ein weiteres Zeichen gelesen, stirbt dieser Verarbeitungsast ab.

Damit können wir den gesuchten Automaten NA_1 formal durch das Quintupel

$$NA_1 = (Q, \Sigma, \delta, q_0, F)$$

beschreiben; dabei sind:

1. Die drei Zustände: $Q = \{q_1, q_2, q_3\}$

2. Das zugrundeliegende Alphabet: $\Sigma = \{0, 1\}$
3. Die Übergangsfunktionen:

	0	1	ϵ
q_1	$\{q_1, q_2\}$	$\{q_1\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\emptyset$

4. Der Startzustand ist q_1.
5. Die Menge der Acceptzustände ist $F = \{q_3\}$.

Das Zustandsdiagramm ist Abbildung [A.1] dargestellt.

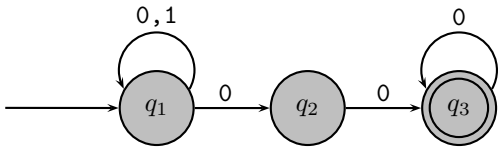


Abbildung A.1: Der nicht-deterministische endliche Automat NA_1.

Um zu verstehen, wie der Automat NA_1 arbeitet, sehen wir uns die Verarbeitung einiger Strings an.

`010100`

Der Automat endet im Acceptzustand q_3 bei der Verarbeitung des Strings `010100`, daher wird dieser akzeptiert.

`11000`

Zu 2 Es ist ein `NFA` zu konstruieren, der die Sprache

$$L_2 = \{w \mid w \text{ enthält den Substring } \texttt{0101}\}$$

mit fünf Zuständen erkennt.

Dieser Automat liest die Zeichenkette ein und verzweigt nichtdeterministische beim Lesen einer `0` vom Startzustand q_s in den Zustand q_0, *erste 0 gelesen.*

Liest er in q_0 eine `1`, geht er von q_0 in den Zustand q_{01} *Teilstring 01* gelesen über. Liest er jedoch in q_0 eine `0`, dann stirbt dieser Zweig ab,

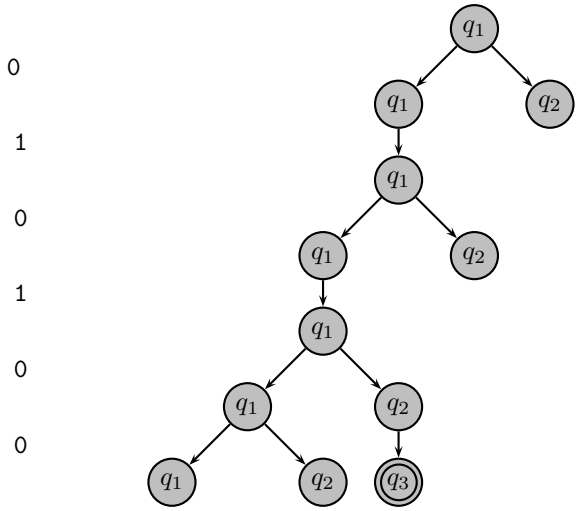


Abbildung A.2: Arbeitsweise des nichtdeterministischen endlichen Automaten NA_1.

da dies nicht die gewünschte Zeichenfolge des Substrings `0101` sein kann. Analog arbeitet der Automat im Zustand q_{01}. Ist das nächste Zeichen eine `0`, geht er in einen Zustand q_{010} über, *Teilstring 010 gelesen*. Ist das folgende Zeichen eine `1` stirbt der Ast ab, i.e. der Zustand q_{01} hat keinen `1` Übergang.

Das gleiche geschieht nochmals im Zustand q_{010}: Wird eine `1` gelesen, ist der komplette Teilstring `0101` vorhanden, die Maschine geht von q_{010} in einen Acceptzustand q_{0101} *Teilstring 0101 gelesen* über. In diesem Zustand können nun beliebig viele weitere Zeichen kommen, der Automat bleibt in q_{0101}, i.e. der q_{0101} Zustand hat also in sich zurücklaufende `0,1` Übergänge. Wird im Zustand q_{010} eine `0` gelesen, stirbt dieser Zweig ab, der q_{010} Zustand hat keinen `1` Übergang.

Das Zustandsdiagramm des nichtdeterministischen endlichen Automaten NA_2 ist in der Abbildung [A.4] dergestellt.

Damit können wir den gesuchten Automaten NA_2 auch formal durch das Quintupel

$$NA_2 = (Q, \Sigma, \delta, q_0, F)$$

beschreiben; dabei ist:

1. Die fünf Zustände: $Q = \{q_s, q_0, q_{01}, q_{010}, q_{0101}\}$
2. Das zugrundeliegende Alphabet: $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
3. Die Übergangsrelation:

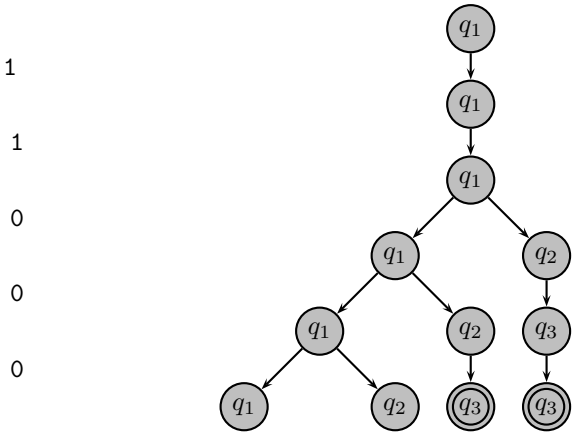


Abbildung A.3: Verarbeitung des Strings 11000.

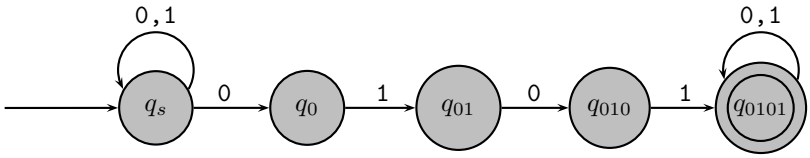


Abbildung A.4: Der nichtdeterministische endliche Automat NA_2.

	0	1	λ
q_s	$\{q_s, q_0\}$	$\{q_s\}$	$\emptyset$
q_0	$\emptyset$	$\{q_{01}\}$	$\emptyset$
q_{01}	$\{q_{010}\}$	$\emptyset$	$\emptyset$
q_{010}	$\emptyset$	$\{q_{0101}\}$	$\emptyset$
q_{0101}	$\{q_{0101}\}$	$\{q_{0101}\}$	$\emptyset$

4. Der Startzustand ist q_s.
5. Die Menge der Acceptzustände ist $F = \{q_{0101}\}$.

Zu 3 Es ist ein NFA zu konstruieren, der die Sprache

$$L_3 = \{w \mid w \text{ enthält eine gerade Anzahl von 0en oder genau zwei 1en}\}$$

erkennt, mit sechs Zuständen. Der gesuchte Automat ist durch das Quintupel

$$NA_3 = (Q, \Sigma, \delta, q_{start}, F)$$

gegeben. Dabei sind:

1. Die Zustandsmenge Q mit den sechs Zuständen:

   $$Q = \{q_s, q^\lambda, q_1, q_{11}, q_g, q_u\}$$

   Dabei ist:

   - q_s ist der Startzustand; der `NFA` verzweigt nichtdeterministisch durch zwei λ Übergänge ohne ein Inputsymbol zu lesen in die beiden Zustände q^λ und q_g.
   - q^λ ist ein Zustand, der dem Szenario entspricht, *noch keine* `1` *gelesen.*
   - q_1 ist ein Zustand, der dem Szenario entspricht, *genau eine* `1` *gelesen.*
   - q_{11} ist ein Zustand, der dem Szenario entspricht, *genau zwei* `1`*en gelesen.*
   - q_g ist ein Zustand, der dem Szenario entspricht, *gerade Anzahl von* `0`*en gelesen*; keine `0` gelesen bedeutet auch eine gerade Anzahl.
   - q_u ist ein Zustand, der dem Szenario entspricht, *ungerade Anzahl von* `0`*en gelesen.*

2. Das Inputalphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$.
3. Der Startzustand $q_{start} = q_s \in Q$.
4. Die Menge der Acceptszustände

   $$F = \{q_{11}, q_g\}.$$

5. Die Übergangsfunktion δ mit:

	`0`	`1`	λ
q_s	$\emptyset$	$\emptyset$	$\{q^\lambda, q_g\}$
q^λ	$\{q^\lambda\}$	$\{q_1\}$	$\emptyset$
q_1	$\{q_1\}$	$\{q_{11}\}$	$\emptyset$
q_{11}	$\{q_{11}\}$	$\emptyset$	$\emptyset$
q_g	$\{q_u\}$	$\{q_g\}$	$\emptyset$
q_u	$\{q_g\}$	$\{q_u\}$	$\emptyset$

Das Zustandsdiagramm dieses Automaten ist in der Abbildung [A.5] dargestellt.

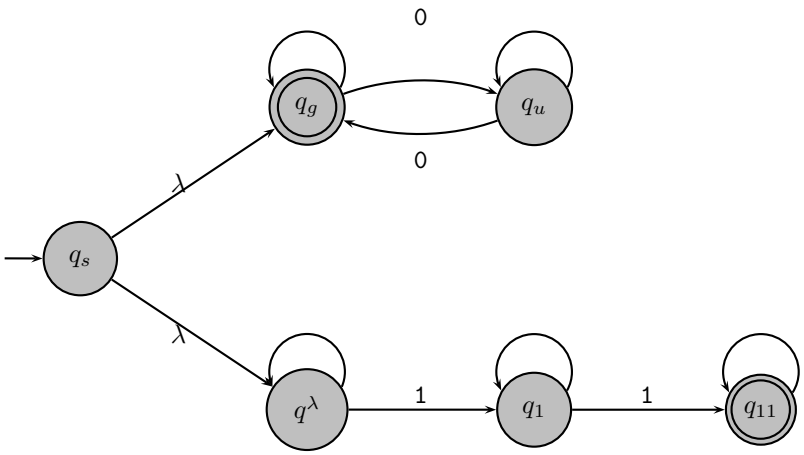


Abbildung A.5: Der nichtdeterministische endliche Automat NA_3.

Die gleiche Sprache wird von dem folgenden `NFA` akzeptiert, der ohne λ–Übergänge arbeitet. Die Menge der Zustände ist:

$$Q = \{q_s, q_0, q_1, p_0, p_1, p_2\}$$

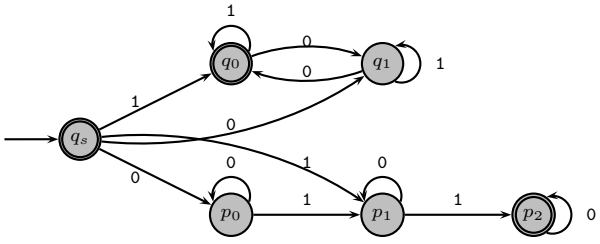
Startzustand ist q_s, die Menge der Acceptzustände ist

$$F = \{q_s, q_0, p_2\},$$

Die Übergangsrelation ist

	0	1
q_s	$\{q_1, p_0\}$	$\{q_0, p_1\}$
q_0	$\{q_1\}$	$\{q_0\}$
q_1	$\{q_0\}$	$\{q_1\}$
p_0	$\{p_0\}$	$\{p_1\}$
p_1	$\{p_1\}$	$\{p_2\}$
p_2	$\{p_2\}$	$\emptyset$

Das Zustandsdiagramm ist:



Zu 4 Es ist ein `NFA` zu konstruieren, der die Sprache

$$L_4 = \{0\}$$

mit zwei Zuständen akzeptiert.

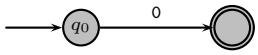
Der folgende `NFA` akzeptiert die Sprache L_4:

$$N = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

mit der Übergangsrelation δ:

	0	1
q_0	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\emptyset$

Das Zustandsdiagramm ist:



Zu 5 Wir konstruieren einen `NFA`, der die Sprache

$$L_5 = 0^*1^*0^*1^*$$

mit vier Zuständen akzeptiert.

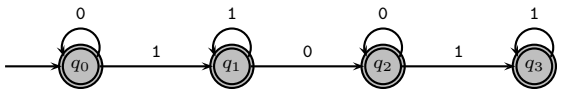
Der folgende `NFA` akzeptiert die Sprache L_5:

$$N = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0, q_1.q_2, q_3\})$$

mit der Übergangsrelation δ:

	0	1
q_0	$\{q_0\}$	$\{q_1\}$
q_1	$\{q_1\}$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_3\}$
q_3	$\emptyset$	$\{q_3\}$

Das Zustandsdiagramm ist:



Zu 6 Es ist ein NFA zu konstruieren, der die Sprache

$$L_6 = \{\lambda\}$$

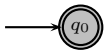
mit einem Zustand akzeptiert.

Der folgende NFA akzeptiert die Sprache L_6:

$$N = (\{q_0\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

mit der Übergangsrelation δ:

$$\delta(q_0, 0) = \delta(q_0, 1) = \emptyset.$$



Zu 7 Wir konstruieren einen NFA, der die Sprache

$$L_7 = 0^*$$

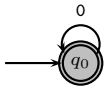
mit einem Zustand akzeptiert.

Der folgende NFA akzeptiert die Sprache L_7:

$$N = (\{q_0\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

mit der Übergangsrelation δ:

$$\delta(q_0, 0) = \{q_0\}, \delta(q_0, 1) = \emptyset.$$



A.3.12 Übung [3.32]

Erstellen Sie die Zustandsdiagramme von deterministischen endlichen Automaten, die die folgenden Sprachen akzeptieren. In allen Fällen ist das zugrundeliegende Alphabet die Menge $\{0, 1\}$:

1. $\{w \mid w$ beginnt mit einer 1 und endet mit einer 0 $\}$.
2. $\{w \mid w$ enthält mindestens drei 1en $\}$.
3. $\{w \mid w$ enthält den Substring 0101, i.e. $w = x0101y$ für x, y $\}$.
4. $\{w \mid w$ hat mindestens die Länge drei und enthält als drittes Symbol die 0 $\}$.
5. $\{w \mid w$ beginnt mit einer 0 und hat ungerade Länge oder beginnt mit einer 1 und hat gerade Länge $\}$.
6. $\{w \mid w$ enthält nicht den Substring 110 $\}$.
7. $\{w \mid$ die Länge von w ist höchstens 5 $\}$.
8. $\{w \mid w$ ist jeder beliebige String mit Ausnahme von 11 und 111 $\}$.
9. $\{w \mid$ jede ungerade Position von w ist eine 1 $\}$.
10. $\{w \mid w$ enthält wenigstens zwei 0en und höchstens eine 1 $\}$.
11. $\{\epsilon, 0\}$.
12. $\{w \mid w$ enthält eine gerade Anzahl von 0en oder genau zwei 1en $\}$.
13. Die leere Menge.
14. Alle Strings mit Ausnahme des leeren Strings.

Lösungen:

Zu 1 In diesem Beispiel ist ein deterministischer endlicher Automat M_1 zu konstruieren, der Strings akzeptiert, die mit einer 1 beginnen und einer 0 enden. Dieser Automat akzeptiert daher Strings der Form 10, 1111110 oder 10000, aber nicht 01, 01111111 usw.

Um die Arbeitsweise solch eines Automaten zu verstehen, spielen wir die Verarbeitung beispielhafter Strings durch.

1. Ist das erste gelesene Zeichen eine 0, dann darf die Maschine niemals in einen Acceptzustand gelangen, unabhängig davon, welche Zeichen dann folgen. Der Automat muß sich also merken, daß er eine 0 gelesen hat. Er geht daher von dem Startzustand q in einen Zustand q_0 über. Egal welche Zeichen nun gelesen werde, er bleibt in diesem Zustand.

2. Ist das erste gelesene Zeichen eine `1`, dann kann dieses Zeichen das erste Zeichen des zu akzeptierenden Strings sein. Ob der String akzeptiert wird, hängt jedoch von den folgenden Zeichen ab. Dies muß sich die Maschine dadurch merken, daß sie von dem Startzustand in einen Zustand q_1 übergeht.
3. Besteht der gelesene String nur aus dem Zeichen `1`, darf sich die Maschine nicht in dem Acceptzustand befinden, mit anderen Worten, der Zustand q_1 kann kein Acceptzustand sein.
4. War das erste Zeichen eine `1`, und folgen ausschließlich `1`en, bleibt der Automat im Zustand q_1.
5. Folgt nach der Verarbeitung von n `1`en eine `0`, dann kann dies ein zu akzeptierender String sein. Dies muß sich der Automat merken, er geht vom Zustand q_1 in einen Zustand q_{10} über, der Acceptzustand ist.
6. Ist der Automat im Zustand q_{10} und liest er jetzt eine `0`, bleibt er in diesem Zustand, denn der gelesene String hat die Form `1x0` und ist ein zu akzeptierender String.
7. Ist der Automat im Zustand q_{10} und liest er eine `1`, kann er nicht im Acceptzustand q_{10} bleiben, da der bisher gelesene String ein nicht zu akzeptierendes Wort sein könnte. Die Maschine muß daher in den Zustand q_1 übergehen.

Damit können wir den Automaten M_1 formal beschreiben durch das Quintupel

$$M_1 = (Q, \Sigma, \delta, q_0, F),$$

mit:

1. Den Zuständen: $Q = \{q, q_0, q_1, q_{10}\}$
2. Das Alphabet: $\Sigma = \{0, 1\}$
3. Die Übergangsfunktionen:

	`0`	`1`
q	q_0	q_1
q_0	q_0	q_0
q_1	q_{10}	q_1
q_{10}	q_{10}	q_1

4. Der Startzustand q^s ist der Zustand q.
5. Die Menge der Acceptzustände ist $F = \{q_{10}\}$.

Zu 2 Es ist ein deterministischer endlicher Automat M_2 zu konstruieren, welcher Worte über dem Alphabet $\Sigma = \{0, 1\}$ akzeptiert, die mindestens drei

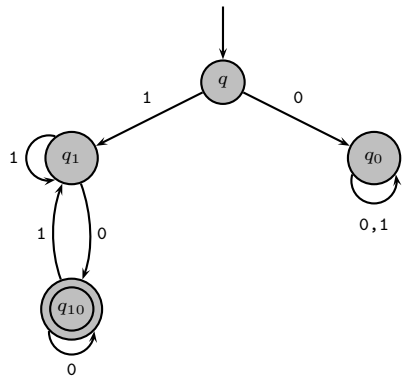


Abbildung A.6: Zustandsdiagramm des Automaten M_1.

1en enthalten. Dabei spielt es keine Rolle, ob diese Zeichen direkt aufeinanderfolgen oder ob 0en dazwischen liegen. Dieser Automat akzeptiert also Strings der Form 111, 000010101010, 1111101010011001 etc. aber nicht 011, 0100000 oder 00100000001.

1. Liest der Automat einen String, der zunächst nur 0en enthält, dann hat der Automat also noch keine 1 verarbeitet; das merkt sich der Automat, er befindet sich im Zustand *noch keine 1 erkannt*, den wir q_0 nennen. Beim Lesen jeder 0 geht der Automat in den gleichen Zustand q_0 über.
2. Liest der Automat eine 1, muß er registrieren, daß die erste der drei 1en verarbeitet wird. Dies realisiert der Automat dadurch, daß er vom Zustand q_0 in den Zustand *erste 1 gelesen* – diesen Zustand nennen wir q_1 – übergeht.
3. Ist die Maschine im Zustand q_1, dann können zwei Fälle auftreten:
   - ↦ Der Automat liest eine 0; dann bleibt er einfach im Zustand q_1.
   - ↦ Der Automat liest eine 1; dann muß er registrieren, daß die zweite 1 verarbeitet wurde. Er geht daher in den Zustand *zweite 1 gelesen* über. Diesen Zustand nennen wir q_{11}.
4. Ist die Maschine im Zustand q_{11}, dann können wiederum zwei Fälle auftreten:
   - ↦ Der Automat liest eine 0; dann bleibt er einfach im Zustand q_{11}, denn er hat ja zwei 1en gelesen.
   - ↦ Der Automat liest eine 1; dann muß er registrieren, daß die dritte 1 verarbeitet wurde. Er geht daher in den Zustand *dritte 1 gelesen* über. Diesen Zustand nennen wir q_{111}.
5. Im Zustand q_{111} spielt es keine Rolle, welche Zeichen nun noch verarbeitet werden, der verarbeitete String enthält drei 1en und ist daher

zu akzeptieren. Daher ist q_{111} ein Acceptzustand, der unter allen Übergängegn in sich übergeht.

Damit können wir den gesuchten Automaten formal durch das Quintupel

$$M_2 = (Q, \Sigma, \delta, q^s, F)$$

beschreiben, mit:

1. Zustände: $Q = \{q_0, q_1, q_{11}, q_{111}\}$
2. Alphabet: $\Sigma = \{\texttt{0}, \texttt{1}\}$
3. Die Übergangsfunktion:

	0	1
q_0	q_0	q_1
q_1	q_1	q_{11}
q_{11}	q_{11}	q_{111}
q_{111}	q_{111}	q_{111}

4. Der Startzustand ist $q^s = q_0$.
5. Die Menge der Acceptzustände $F = \{q_{111}\}$.

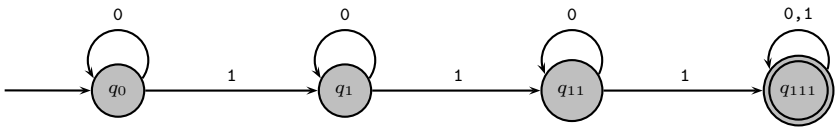


Abbildung A.7: Zustandsdiagramm des Automaten M_2.

Zu 3 Es ist ein deterministischer endlicher Automat M_3 zu konstruieren, der Worte akzeptiert, die den Substring `0101` enthalten. Dieser Automat akzeptiert also Strings der Form `0101`, `0000 0101 0000000`, `11111 0101 0011001` etc. aber nicht `111`, `0011` usw.

Wenn dieser Automat irgend einen Inputstring liest, der zunächst nur `1`en enthält — *e.g.* `11111 0101` ... — dann werden alle diese `1`en ignoriert, da diese nicht das erste Symbol des Teilstrings darstellen können. Der Automat befindet sich nach Verabeitung der `1`en in einem Zustand *noch kein Zeichen des Musters erkannt* den wir q nennen.

Wenn bei der Verarbeitung nun die erste `0` auftritt, kann dieses Symbol das erste Zeichen des gesuchten Musters sein. Diesen Fakt muß der Automat registrieren. Der Automat geht also in einen Zustand *erstes Zeichen des Teilstrings gelesen* über, den wir q_0 nennen.

Wenn sich der Automat im Zustand q_0 befindet und das nächste Inputsymbol verarbeitet, können zwei Möglichkeiten auftreten:

- Das nächste Zeichen, das gelesen wird, ist wieder eine `0`. Dann war die vorhergehende `0` nicht das erste Zeichen des gesuchten Substrings. Die gerade verarbeitete `0` könnte aber die erste `0` des Musters sein, daher muß der Automat in dem Zustand q_0 bleiben.
- Das nächste Zeichen, das gelesen wird, ist eine `1`. Dies kann das zweite Zeichen des Musters sein, diese Tatsache, muß sich der Automat merken. Der Automat heht daher von dem Zustand q_0 in den Zustand q_{01} über.

Abhängig von dem nächsten Zeichen, treten nun wieder zwei Fälle auf.

- Der Automat ist im Zustand q_{01} und liest eine `1`. Der bisher gelesene String hat also die Form `11...  1  011`. Dann muß der Automat in den Zustand *noch kein Zeichen des Teilstrings gelesen* übergehen, das ist der Zustand q.
- Der Automat ist im Zustand q_{01} und liest eine `0`. Dann hat er also den Teilstring `010` gelesen, eine Tatsache, die sich der Automat merken muß, er geht daher in den Zustand q_{010} über.

Der Automat ist im Zustand q_{010} und verarbeitet das nächste Inputzeichen. Auch hier können zwei Fälle auftreten:

- Der Automat liest eine `0`. Die letzten vier verarbeitete Zeichen sind also `0100`. Daher muß der Automat wieder in den Zustand übergehen, den wir *das erste Zeichen des Musters erkannt* genannt haben, denn der `0100` String ist nicht der gesuchte Teilstring, aber die gerade gelesene `0` könnte das erste Zeichen des Musters sein. Damit muß der Automat nach Lesen der `0` vom Zustand q_{010} in den Zustand q_0 übergehen.
- Es wird eine `1` gelesen. Die letzten vier Zeichen haben also genau die Form des gesuchten Musters `0101`. Diese Tatsache wird von dem Automaten dahingehend registriert, daß er in den Zustand q_{0101} übergeht, dies muß der Acceptzustand sein.

Bei jedem weiteren gelesenen Inputzeichen geht der Zustand q_{0101} in sich über.

Damit sind wir in der Lage, den gesuchten Automaten formal durch das Quintupel

$$M_3 = (Q, \Sigma, \delta, q_0, F)$$

zu beschreiben, mit:

1. Zustände: $Q = \{q, q_0, q_{01}, q_{010}, q_{0101}\}$

2. Alphabet: $\Sigma = \{0, 1\}$
3. Die Übergangsfunktionen:

	0	1
q	q_0	q
q_0	q_0	q_{01}
q_{01}	q_{010}	q
q_{010}	q_0	q_{0101}
q_{0101}	q_{0101}	q_{0101}

4. Der Startzustand $q^s = q$.
5. Die Menge der Acceptzustände $F = \{q_{0101}\}$.

Das Zustandsdiagramm des Automaten M_3 ist in der Abbildung [A.8] dargestellt.

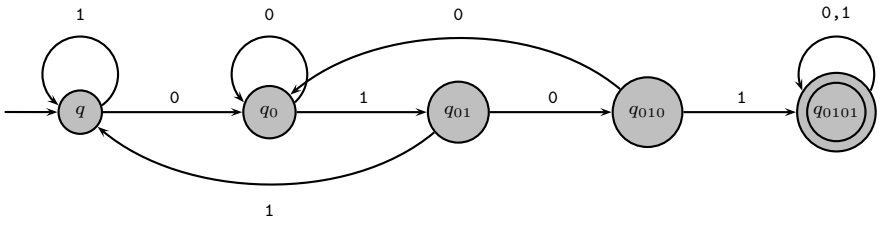


Abbildung A.8: Zustandsdiagramm des Automaten M_3.

Zu 4 In diesem Beispiel ist ein endlicher deterministischer Automat zu konstruieren, der Worte über dem Alphabet $\Sigma = \{0, 1\}$ akzeptiert, die mindestens die Länge drei haben und als drittes Symbol das Zeichen 0 enthält. Dieser Automat akzeptiert also Strings der Form 1101111, 0101, 01011011 etc.

Wenn der Automat einen Eingabestring verarbeitet, spielt es keine Rolle, ob die ersten beiden Zeichen eine 0 oder eine 1 ist. Der Automat muß sich aber merken, daß zwei Zeichen gelesen wurden. Je nachdem, welches Zeichen dann gelesen wird, können zwei Möglichkeiten auftreten:

- Der Automat liest als drittes Zeichen eine 1. Dann darf er niemals in einem Acceptzustand enden, unabhängig davon, welche Zeichen noch gelesen werden.
- Der Automat liest als drittes Zeichen eine 0. Dann muss er in einen Acceptzustand übergehen, auch dies unabhängig davon, welche Zeichen noch gelesen werden.

Damit sind wir in der Lage, den gesuchten Automaten formal durch das Quintupel

$$M_4 = (Q, \Sigma, \delta, q_0, F)$$

zu beschreiben, mit:

1. Zustände: $Q = \{q_0, q_1, q_2, q_3, q_4\}$
2. Alphabet: $\Sigma = \{\texttt{0}, \texttt{1}\}$
3. Die Übergangsfunktion δ:

	0	1
q_0	q_1	q_1
q_1	q_2	q_2
q_2	q_3	q_4
q_3	q_3	q_3
q_4	q_4	q_4

4. Der Startzustand $q^s = q_0$.
5. Die Menge der Acceptzustände ist $F = \{q_4\}$.

Das Zustandsdiagramm des Automaten M_4 ist in Abbildung [A.9] dargestellt.

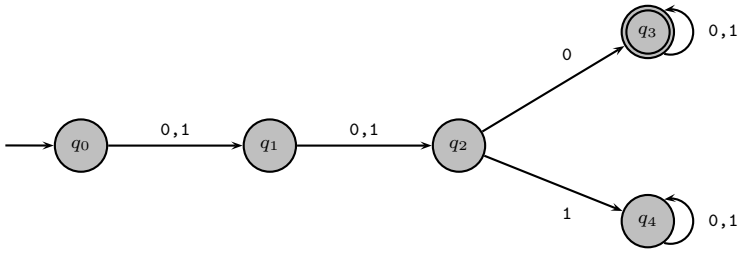


Abbildung A.9: Zustandsdiagramm des Automaten M_4.

Zu 5 In diesem Beispiel ist ein endlicher deterministischer Automat zu konstruieren, der Worte über dem Alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$ akzeptiert, die entweder mit dem Zeichen `0` beginnen und eine ungerade Anzahl von Zeichen haben oder mit dem Zeichen `1` beginnen und eine gerade Anzahl von Zeichen haben. Dieser Automat akzeptiert also Strings der Form `110111`, `010`, `00000` etc.

Wenn dieser Automat sich im Startzustand q_s befindet und das erste Zeichen des Inputstrings liest, treten zwei Möglichkeiten auf:

- Der Automat liest eine `0`. Dies muß der Automat registrieren, indem er in einen Zustand *erstes Zeichen ist eine 0* übergeht, den wir q_0 nennen. Hier können nun drei Möglichkeiten auftreten:

  - ↦ Es folgt kein weiteres Zeichen. Dann muß sich der Automat in einem Acceptzustand befinden, denn er hat ein Zeichen (die `0`) gelesen und dies ist eine ungerade Anzahl von Zeichen. Daher ist q_0 Acceptzustand.
  - ↦ Es folgt nun eine gerade Anzahl von Zeichen, dann hat er insgesamt eine ungerade Anzahl von Inputsymbolen verarbeitet und er muß sich in einem Acceptzustand befinden.
  - ↦ Es folgt nun eine ungerade Anzahl von Zeichen, dann hat er insgesamt eine gerade Anzahl von Inputsymbolen verarbeitet und er darf sich nicht in einem Acceptzustand befinden.
- Der Automat liest eine `1`. Dies muß der Automat ebenfalls registrieren, indem er in einen Zustand *erstes Zeichen ist eine 1* übergeht, diesen Zustand nennen wir q_1. Hier können nun ebenfalls drei Fälle auftreten:
  - ↦ Es folgt kein weiteres Zeichen. Dann darf sich der Automat nicht in einem Acceptzustand befinden, denn er hat ein Zeichen (die `1`) gelesen und dies ist eine ungerade Anzahl von Zeichen. Daher ist q_1 kein Acceptzustand.
  - ↦ Es folgt nun eine ungerade Anzahl von Zeichen, dann hat er insgesamt eine gerade Anzahl von Inputsymbolen verarbeitet und er muß sich in einem Acceptzustand befinden.
  - ↦ Es folgt nun eine gerade Anzahl von Zeichen, dann hat er insgesamt eine ungerade Anzahl von Inputsymbolen verarbeitet und er darf sich nicht in einem Acceptzustand befinden.

Damit können wir den gesuchten Automaten M_e formal durch das Quintupel

$$M_5 = (Q, \Sigma, \delta, q_0, F)$$

beschreiben; dabei ist:

1. Die fünf Zustände: $Q = \{q_s, q_0, q_1, q_0^g, q_1^u\}$
2. Das zugrundeliegende Alphabet ist $\Sigma = \{\texttt{0}, \texttt{1}\}$
3. Die Übergangsfunktion ist:

	`0`	`1`
q_s	q_0	q_1
q_0	q_0^g	q_0^g
q_1	q_1^u	q_1^u
q_0^g	q_0	q_0
q_1^u	q_1	q_1

4. Der Startzustand ist q_s.
5. Die Menge der Acceptzustände $F = \{q_0, q_1^u\}$.

Das Zustandsdiagramm des Automaten M_5 ist in Abbildung [A.10] dargestellt.

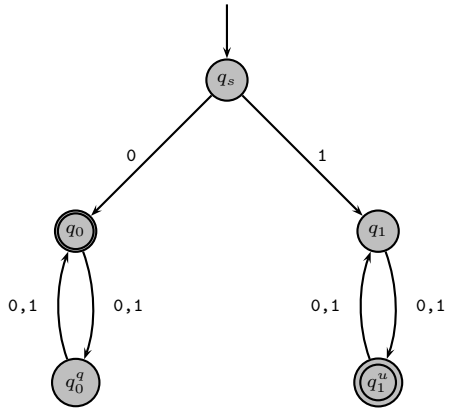


Abbildung A.10: Zustandsdiagramm des Automaten M_5.

Zu 6 In diesem Beispiel ist ein endlicher deterministischer Automat zu konstruieren, der alle Worte über dem Alphabet $\Sigma = \{0, 1\}$ akzeptiert, die nicht den Substring 110 enthalten.

Die zu konstruierende Maschine muss also alle Strings akzeptieren, die *nicht* den 110 Substring enthalten. Angenommen der zu lesende String enthält zunächst nur 0en. Dies braucht sich die Maschine nicht zu merken. Sie bleibt daher in einem Zustand, den wir *noch keine 1 des Teilstrings gelesen* nennen und bezeichnen diesen Zustand mit q_s. Dies ist auch gleichzeitig der Startzustand. Der Zustand q_s ist auch Acceptzustand, da beispielsweise der String 00 ein akzeptiertes Wort sein muß.

Wird nun eine 1 gelesen, muß sich der Automat dies merken, denn es könnte die erste 1 des nicht erwünschten Teilstrings sein. Der Automat geht dann in einen Zustand *erste 1 gelesen* über, den wir q_1 nennen. Auch dieser Zustand ist ein Acceptzustand, denn Strings der Form 000...01 sind zu akzeptierende Wörter über $\Sigma = \{0, 1\}$.

Ist der Automat in dem Zustand *erste 1 gelesen*, muß er beim Verarbeiten des folgenden Zeichens zwei Fälle unterscheiden können:

- ⇝ Ist das Zeichen eine 0, dann geht er in den Zustand *noch keine 1 des Teilstrings gelesen* zurück.
- ⇝ Liest er eine 1, dann könnte dies die zweite 1 des Substrings sein. Dies muß registriert werden, der Automat geht in einen Zustand *zweite 1*

des Teilstrings gelesen über; diesen Zustand nennen wir q_{11}. Auch dieser Zustand ist ein Acceptzustand.

Ist der Automat im Zustand q_{11}, dann müssen beim Lesen des folgenden Zeichens erneut zwei Fälle unterschieden werden:

- ⇝ Ist das Zeichen eine `0`, dann ist der unerwünschte Teilstring `110` gelesen. Dies muß registriert werden, er geht in einen Zustand *Teilstring gelesen* über, den wir q_{110} nennen. Dieser Zustand ist kein Acceptzustand, alle weiteren Zeichen, die eventuell noch gelesen werden, führen diesen Zustand in sich selbst über.
- ⇝ Liest er eine `1`, dann könnte diese `1` auch die zweite `1` des Substrings sein. Daher bleibt der Automat beim Lesen einer `1` in dem Zustand *zweite 1 des Teilstrings gelesen*.

Damit können wir den gesuchten Automaten M_6 formal durch das Quintupel

$$M_6 = (Q, \Sigma, \delta, q_0, F)$$

beschreiben; dabei ist:

1. Die vier Zustände: $Q = \{q_s, q_1, q_{11}, q_{110}\}$
2. Das zugrundeliegende Alphabet: $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
3. Die Übergangsfunktion δ :

	`0`	`1`
q_s	q_s	q_1
q_1	q_s	q_{11}
q_{11}	q_{110}	q_{11}
q_{110}	q_{110}	q_{110}

4. Der Startzustand ist q_s.
5. Die Menge der Acceptzustände ist $F = \{q_s, q_1, q_{11}\}$.

Das Zustandsdiagramm des Automaten M_f ist in Abbildung [A.11] dargestellt.

Zu 7 In diesem Beispiel ist ein endlicher deterministischer Automat zu konstruieren, der alle Worte über dem Alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ akzeptiert, die die Länge kleiner gleich 5 haben. Der Automat hat also alle Strings der Form `0,1,00,01,10,...11111` zu akzeptieren, längere Zeichenketten dürfen den Automaten nicht in einen Acceptzustand bringen.

Liest der Automat das erste Zeichen eines Inputstrings, dann spielt es keine Rolle, welches Zeichen verarbeitet wird. Der Automat muß jedoch

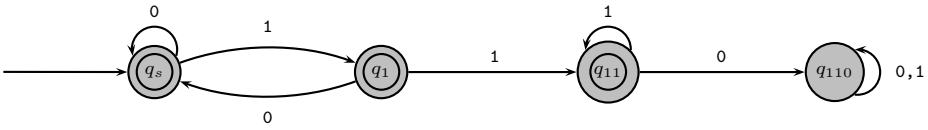


Abbildung A.11: Zustandsdiagramm des Automaten M_6.

registrieren, daß ein Zeichen gelesen wurde. Diesen Zustand, *erstes Zeichen gelesen*, nennen wir q_1. Dies muß auch gleichzeitig ein Acceptzustand sein, da der Inputstring auch aus einem einzelnen Zeichen bestehen kann, dessen Länge ist sicher kleiner als 5, daher muß die Maschine solch einen String akzeptieren.

Analoges gilt für das Lesen des zweiten Zeichens. Es spielt keien Rolle, ob eine `0` oder eine `1` gelesen wird, der Automat muß nur registrieren, es wurde das zweite Zeichen gelesen. Daher geht er vom Zustand q_1 in den Zustand *zweites Zeichen gelesen* über, den wir q_2 nennen, dieser ist ebenfalls ein Acceptzustand.

Dies geht so weiter bis zum Lesen des fünften Zeichens. Dann befindet sich der automat im Zustand *fünftes Zeichen gelesen*, dieser Zustand heißt q_5 und ist Acceptzustand. Ist der String nun beendet, hat der Automat den String akzeptiert, denn er befindet sich in einem Acceptzustand. Folgen jedoch weitere Zeichen, dann muß der Automat in einen zustand übergehen, den wir *mehr als fünf Zeichen gelesen* nennen, und mit $q_{>5}$ bezeichnen. Unabhängig davon, wieviele und welche Zeichen gelesen werden, verbleibt die Maschine in diesem Zustand, dieser ist kein Acceptzustand.

Damit können wir den gesuchten Automaten M_g formal durch das Quintupel

$$M_7 = (Q, \Sigma, \delta, q_0, F)$$

beschreiben; dabei ist:

1. Die sieben Zustände: $Q = \{q_s, q_1, q_2, q_3, q_4, q_5, q_{>5}\}$
2. Das zugrundeliegende Alphabet: $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
3. Die Übergangsfunktion:

	0	1
q_s	q_1	q_1
q_1	q_2	q_2
q_2	q_3	q_3
q_3	q_4	q_4
q_4	q_5	q_5
q_5	$q_{>5}$	$q_{>5}$
$q_{>5}$	$q_{>5}$	$q_{>5}$

4. Der Startzustand ist q_s.
5. Die Menge der Acceptzustände $F = \{q_1, q_2, q_3, q_4, q_5\}$.

Das Zustandsdiagramm des Automaten M_7 ist in der Abbildung [A.12] dargestellt.

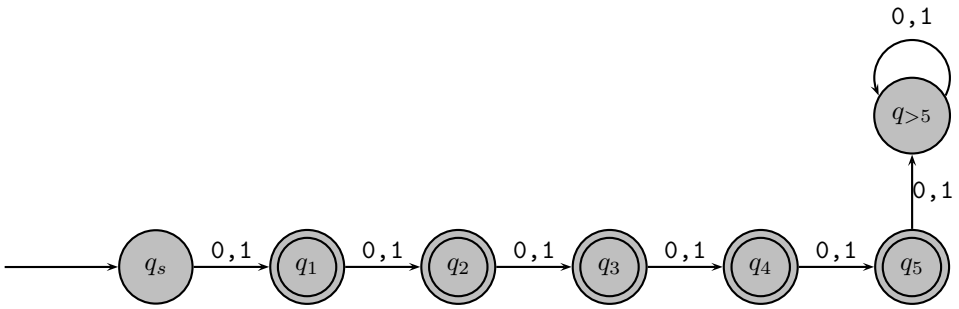


Abbildung A.12: Das Zustandsdiagramm des Automaten M_7.

Zu 8 In diesem Beispiel ist ein endlicher deterministischer Automat zu konstruieren, der alle Worte über $\Sigma = \{0, 1\}$ akzeptiert mit Ausnahme der beiden Strings 11 und 111.

Wenn dieser Automat einen String verarbeitet, dessen erstes Symbol eine 0 ist, dann muß er – unabhängig davon welche und wieviele andere Symbole folgen – diesen String akzeptieren. Es geht daher in einen Zustand über, den man *String ist zu akzeptiern*, q_f nennen kann. Dies muß als ein Acceptzustand sein.

Ist das erste Symbol eine 1, dann könnte dies die erste 1 der nicht zu akzeptierenden Strings 11 oder 111 sein. Der Automat muß dies registrieren. Er geht daher vom Startzustand in einen Zustand *erste 1 gelesen* über. Dies

muß aber auch ein Acceptzustand sein, da der aus dem einzigen Symbol `1` bestehende String zu akzeptieren ist. Diesen Zustand nennen wir q_1.

Befindet sich der Automat im Zustand q_1, dann treten zwei Fälle ein:

- ⇝ Es wird eine `0` gelesen. Dann ist dieser String zu akzeptieren, unabhängig davon, welche und wieviele Zeichen noch folgen. Der Automat geht daher in den oben erwähnten Zustand q_f über.
- ⇝ Es wird eine `1` gelesen. Damit wurde die zweite `1` gelesen; folgt nun kein weiteres Inputsymbol, ist dieser String nicht zu akzeptieren. Dies muß sich der Automat wieder merken, er geht beim Lesen der `1` daher vom Zustand q_1 in einen Zustand *zweite 1 gelesen* über, den wir q_{11} nennen.

Im q_{11} Zustand gilt gleiches wie im q_1 Zustand. ist nun eine `0` zu verarbeiten, geht der Automat in den Zustand q_f über, beim Lesen der `1` in einen Zustand *dritte 1 gelesen*, der q_{111} heißt.

Im Zustand q_{111} spielt es keine Rolle, ob nun eine `0` oder `1` verarbeitet wird, wenn weitere Symbole folgen. Es muß ein `0,1`-Übergang von q_{111} nach q_f vorhanden sein.

Damit können wir den gesuchten Automaten M_h formal durch das Quintupel

$$M_8 = (Q, \Sigma, \delta, q_0, F)$$

beschreiben; dabei ist:

1. Die fünf Zustände: $Q = \{q_s, q_1, q_{11}, q_{111}, q_f\}$
2. Das zugrundeliegende Alphabet: $\Sigma = \{\texttt{0}, \texttt{1}\}$
3. Die Übergangsfunktion δ:

	`0`	`1`
q_s	q_f	q_1
q_1	q_f	q_{11}
q_{11}	q_f	q_{111}
q_{111}	q_f	q_f
q_f	q_f	q_f

4. Der Startzustand ist q_s.
5. Die Menge der Acceptzustände ist $F = \{q_1, q_f\}$.

Das Zustandsdiagramm diese Automaten ist in Abbildung [A.13] dargestellt.

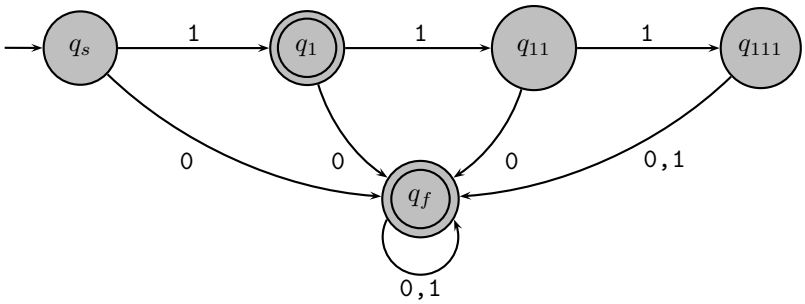


Abbildung A.13: Das Zustandsdiagramm des Automaten M_8.

Zu 9 In diesem Beispiel ist ein endlicher deterministischer Automat zu konstruieren, der alle Worte der Sprache

$$\{w \mid \text{ jede ungerade Position von } w \text{ ist eine } \texttt{1} \}$$

akzeptiert. Dieser Automat muss Strings wie zum Beispiel `1`, `10`, `11`, `101`, `111` oder `11111` usw. akzeptieren, aber nicht Strings wie `0...`, `10100` etc.

Liest der Automat daher das erste Zeichen, muss sofort unterschieden werden, ob es eine `0` oder `1` ist.

- ⇝ Es wird eine `0` gelesen. Dann ist dieser String in jedem Fall nicht zu akzeptieren, unabhängig davon, welche und wieviele Zeichen noch folgen. Da die erste Position eine ungerade Zahl ist, und dort eine `0` steht, sind solche Strings nicht zu akzeptieren. Der Automat geht daher vom Startzustand in einen Zustand q_0 über. Egal, welche Zeichen nun folgen, q_0 geht in sich über.
- ⇝ Es wird eine `1` gelesen. Folgt nun kein weiteres Inputsymbol, ist dieser String nicht zu akzeptieren. Dies muss sich der Automat merken, er geht beim Lesen der `1` daher vom Startzustand q_s in einen Zustand *erste 1 an ungerader Position gelesen* über, den wir q_1 nennen. Dies muss ein Acceptzustand sein, da der String `1` zu akzeptieren ist.

Ist der Automat im Zustand q_1 und liest er ein weiteres Zeichen, dann ist dieser String zu akzeptieren, unabhängig davon, welches Zeichen gelesen wird. Dies sind die Strings `10` und `11`. Er geht nun in einen Zustand q_{1x} über. Dieser Zustand akzeptiert Strings mit einer `1` an ungerader Position und einem beliebigen Zeichen an der geraden Position.

Die Übergänge aus Zustand q_{1x} hängen nun davon ab, welche Zeichen gelesen werden.

- ⇝ Es wird eine `0` gelesen. Dann ist dieser String nicht zu akzeptieren, unabhängig davon, welche und wieviele Zeichen noch folgen, denn dieser String hat auf jeden Fall an der dritten Position eine `0`. Der Automat geht daher in den oben erwähnten Zustand q_0 über.
- ⇝ Es wird eine `1` gelesen. Damit wurde die zweite `1` an einer ungeraden Position gelesen; folgt nun kein weiteres Inputsymbol, ist dieser String zu akzeptieren. Der Automat geht beim Lesen der `1` daher vom Zustand q_{1x} in den Zustand q_1 über.

Damit können wir den gesuchten Automaten M_i formal durch das Quintupel

$$M_9 = (Q, \Sigma, \delta, q^s, F)$$

beschreiben; dabei ist:

1. Die vier Zustände: $Q = \{q_s, q_1, q_{1x}, q_0\}$
2. Das zugrundeliegende Alphabet: $\Sigma = \{\texttt{0}, \texttt{1}\}$
3. Die Übergangsfunktion δ:

	`0`	`1`
q_s	q_0	q_1
q_1	q_{1x}	q_{1x}
q_{1x}	q_0	q_1
q_0	q_0	q_0

4. Der Startzustand ist q_s.
5. Die Menge der Acceptzustände ist $F = \{q_1, q_{1x}\}$.

Das Zustandsdiagramm diese Automaten ist in Abbildung [A.14] dargestellt.

Zu 10 In diesem Beispiel ist ein endlicher deterministischer Automat zu konstruieren, der alle Worte der Sprache

$$L = \{w \mid \text{ enthält mindestens zwei } \texttt{0}\text{en und höchstens eine } \texttt{1} \}$$

akzeptiert. Dieser Automat muss Strings wie zum Beispiel `x001`, `100x`, `x010x` akzeptieren, wobei `x` für eine beliebige Anzahl von `0`en steht. Er akzeptiert auch alle Strings, die nur aus `0`en bestehen, denn höchstens eine `1` bedeutet, keine `1` oder eine `1`.

Liest der Automat das erste Zeichen, treten zwei Unterscheidungen auf, die der Automat registrieren muss:

- ⇝ Liest er eine `0`, ist dies die erste der mindestens vorhandenen beiden `0`en. Der Automat geht in einen Zustand *erste 0 gelesen* über, den wir q_0 nennen.

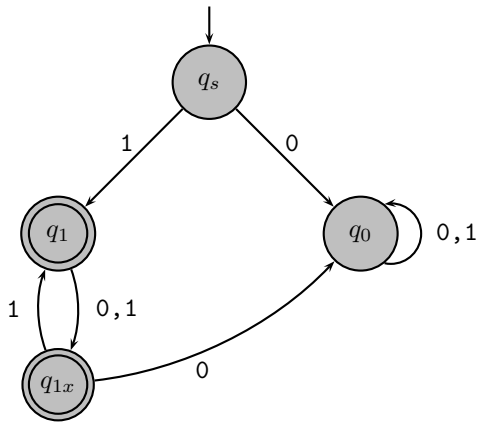


Abbildung A.14: Zustandsdiagramm des deterministischen endlichen Automaten M_9.

⇝ Liest er eine `1`, ist dies die einzige `1`, die in einem String vorhanden sein darf, damit dieser noch akzeptiert wird. Er geht vom Startzustand q_s in den Zustand *die* `1` *gelesen* über, den wir q_1 nennen.

Jetzt wird das zweite Zeichen gelesen. Wir verfolgen zunächst den Ablauf im Zustand q_1.

- Ist er im Zustand q_1 und wird eine `0` gelesen, muß er registrieren, daß die erste `0` verarbeitet wurde. Der Automat geht daher in einen Zustand *Zeichenkette 10 gelesen* über, dies ist der Zustand q_{10}.
- Ist er im Zustand q_1 und liest er eine `1`, ist dies die zweite gelesene `1`, dieser String ist – unabhängig davon, welche Zeichen noch folgen – nicht zu akzeptieren. Der Automat geht daher vom Zustand q_1 Zustand *String reject* über, diesen nennen wir q_r. q_r hat in sich übergehende Zustände `0,1`, denn befindet sich der Automat einmal in diesem Zustand, dann verbleibt er darin, da alle String unakzeptabel sind, die den Automat in q_r bringen.

Liest er in q_{10} das dritte Zeichen und dieses ist eine `1`, geht er ebenfalls in den Zustand q_r über. Ist das dritte Zeichen eine `0`, dann muss der Automat dies registrieren, denn er hat — folgt jetzt beispielsweise kein weiteres Zeichen — eventuell einen String verarbeitet, der akzeptiert werden muss. Es gibt daher einen Übergang $q_{10} \xrightarrow{0} q_{100}$. Dieser Zustand q_{100} muss ein Acceptzustand sein. Werden jetzt weitere `0`en gelesen, verbleibt der Automat im Zustand q_{100}. Wird jedoch im Zustand q_{100} eine `1` gelesen, dann sind mindestens zwei `0`en gelesen, aber mehr als eine `1`. Solch ein String

wird nicht akzeptiert, daher führt der Automat den Übergang $q_{100} \xrightarrow{\texttt{1}} q_r$ aus. Damit ist der Teilgraph für q_1 abgeschlossen.

Als nächstes untersuchen wir das Szenario, dass sich der Automat im Zustand q_0 befindet und das zweite Zeichen liest.

- $\rightsquigarrow$ Liest er eine `0`, ist dies die zweite der mindestens vorhandenen beiden `0`en. Der Automat geht in einen Zustand *zweite 0 gelesen* über, den wir q_{00} nennen.
- $\rightsquigarrow$ Liest er eine `1`, könnte dies die einzige `1` sein, die in einem String vorhanden sein darf, damit dieser noch akzeptiert wird. Er geht daher vom Zustand q_0 in den Zustand *den Teilstring* `01` *gelesen* über, den wir q_{01} nennen.

Zwischen den Zuständen q_{00} und q_{01} muß differenziert werden, da der Automat nun unterscheiden muß, was beim Lesen des nächsten Zeichens geschehen soll. Ist er im Zustand q_{00} und folgen nun ausschließlich `0`en, bleibt er solange in q_{00} bis die erste `1` liest. Dann muß er in einen Acceptzustand übergehen, dies erreichen wir durch den Übergang $q_{00} \xrightarrow{\texttt{1}} q_{100}$.

Ist er im Zustand q_{01} und liest er jetzt eine `1`, muß der Automat in den q_r Zustand übergehen, da mehr als eine `1` gelesen wurden. Liest er aber eine `0`, geht er von q_{01} in den Acceptzustand q_{100} über.

Damit ist der Automat vollständig beschrieben.

Damit können wir den gesuchten Automaten M_{10} formal durch das Quintupel

$$M_{10} = (Q, \Sigma, \delta, q^s, F)$$

beschreiben; dabei ist:

1. Die acht Zustände: $Q = \{q_s, q_0, q_1, q_{00}, q_{01}, q_{10}, q_{100}, q_r\}$
2. Das zugrundeliegende Alphabet: $\Sigma = \{\texttt{0}, \texttt{1}\}$
3. Die Übergangsfunktion δ:

	`0`	`1`
q_s	q_0	q_1
q_0	q_{00}	q_{01}
q_1	q_{10}	q_r
q_{00}	q_{00}	q_{100}
q_{01}	q_{100}	q_r
q_{10}	q_{100}	q_r
q_{100}	q_{100}	q_r
q_r	q_r	q_r

4. Der Startzustand ist q_s.

5. Die Menge der Acceptzustände ist $F = \{, q_{00}, q_{100}\}$.

Das Zustandsdiagramm diese Automaten ist in Abbildung [A.15] dargestellt.

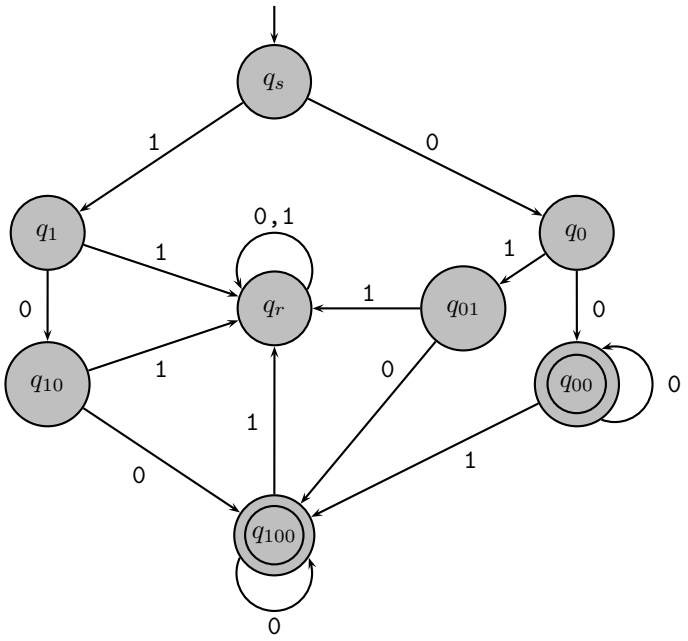


Abbildung A.15: Der deterministische endliche Automat M_{10}.

An diesem Beispiel sind deutlich zwei wichtige Aspekte bei der Konstruktion endlicher deterministischer Automaten zu erkennen:

1. Endliche deterministische Automaten sind nicht eindeutig.
2. Zustände mit gleichen Übergängen sind redundant.

Wie aus der Abbildung [A.15] des Zustandsdiagramms des Automaten M_j zu erkennen ist, haben die beiden Zustände q_{01} und q_{10} die gleichen Übergänge:

$$q_{01} \xrightarrow{0} q_{100} \qquad q_{01} \xrightarrow{1} q_r$$
$$q_{10} \xrightarrow{0} q_{100} \qquad q_{10} \xrightarrow{1} q_r$$

Das bedeutet, der in der Abbildung [A.15] dargestellte Automat M_j ist äquivalent zu einem deterministischen endlichen Automaten mit

$$M'_{10} = (Q, \Sigma, \delta, q_0, F)$$

1. Sieben Zustände: $Q = \{q_s, q_0, q_1, q_{00}, q_{01}, q_{100}, q_r\}$
2. Das zugrundeliegende Alphabet: $\Sigma = \{0, 1\}$
3. Die Übergangsfunktion:

	0	1
q_s	q_0	q_1
q_0	q_{00}	q_{10}
q_1	q_{10}	q_r
q_{00}	q_{00}	q_{100}
q_{01}	q_{100}	q_r
q_{100}	q_{100}	q_r
q_r	q_r	q_r

4. Der Startzustand ist q_s.
5. Die Menge der Acceptzustände ist $F = \{q_{100}\}$.

Äquivalent bedeutet hier, dass die Automaten M_{10} nd M'_{10} die gleiche Sprache erkennen.

Das Zustandsdiagramm des Automaten M'_{10} ist in der Abbildung [A.16] dargestellt.

Zu 11 Der zu konstruierende Automat hat die Sprache $\{\lambda, 0\}$ zu erkennen. Er akzeptiert also den leeren String λ und den String `0`, der aus genau einem Symbol besteht.

Damit der Automat den leeren String akzeptiert, ist der Startzustand auch ein Acceptzustand. Liest der Automat nun das erste Zeichen, geht er beim Lesen einer `0` in den Acceptzustand *0 gelesen* über, den wir q_0 nennen. Liest er eine `1`, ist der String nicht zu akzeptieren, der Automat geht in einen Zustand *reject* über, den wir q_r nennen. Ist er einmal in diesem Zustand, dann bleibt er auch in q_r unabhängig davon, welche und wieviele Zeichen noch gelesen werden. Ist er im Zustand q_0 und liest er irgend ein weiteres Zeichen, muss er ebenfalls in den reject Zustand q_r übergehen.

Damit ist der Automat M_{11} formal beschreibbar durch

$$M_{11} = (Q, \Sigma, \delta, q_0, F)$$

1. Mit drei Zuständen: $Q = \{q_s, q_0, q_r\}$
2. Das zugrundeliegende Alphabet: $\Sigma = \{0, 1\}$
3. Die Übergangsfunktion δ:

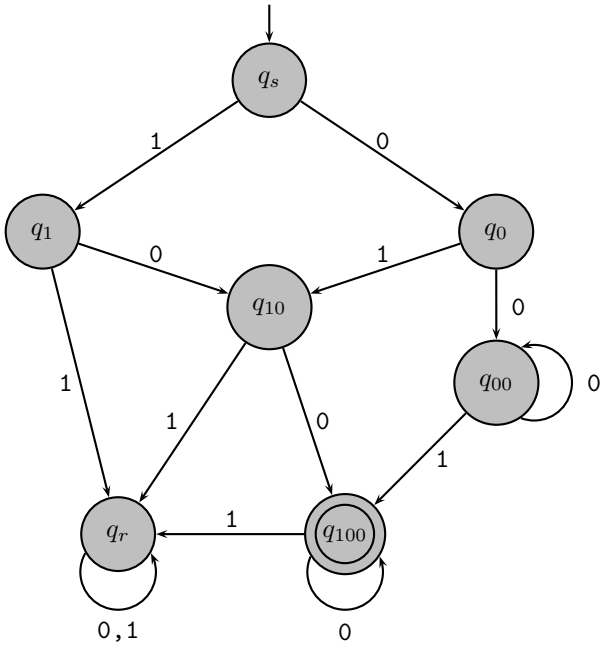


Abbildung A.16: Der deterministische endliche Automat M'_{10}.

	0	1
q_s	q_0	q_r
q_0	q_r	q_r
q_r	q_r	q_r

4. Der Startzustand ist q_s.
5. Die Menge der Acceptzustände ist $F = \{q_s, q_0\}$.

Zu 12 Es ist ein deterministischer endlicher Automat zu konstruieren, der die Sprache

$$L = \{w \mid w \text{ enthält eine gerade Anzahl von } \texttt{0}\text{en} \\ \text{oder genau zwei } \texttt{1}\text{en}\}$$

erkennt.

Dieser Automat muss also Strings wie `01101`, `10000` ..., erkennen, *i.e.* die Strings *enthalten* eine gerade Anzahl von `0`en und eine beliebige Zahl — außer zwei — von `1`en oder die Strings enthalten genau zwei `1`en, dann können beliebig viele `0`en auftreten.

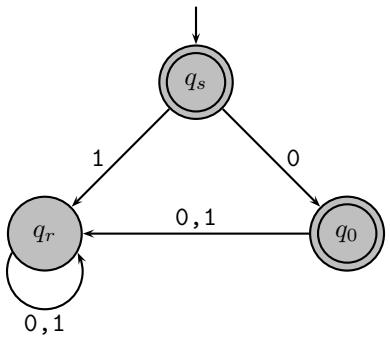


Abbildung A.17: Der deterministische endliche Automat M_{11}, der die Sprache $\{\lambda, 0\}$ erkennt.

Der Automat muss also zählen, ob er bereits zwei 1en gelesen hat und er muss immer registrieren, ob eine gerade oder eine ungerade Anzahl von 0en gelesen wurde. Wir definieren zunächst folgende Zustände:

- $q^g_{0>1}$: Zustand, gerade Anzahl von 0en und mehr als zwei 1en gelesen.
- $q^u_{0>1}$: Zustand, ungerade Anzahl von 0en und mehr als zwei 1en gelesen.

Der Automat befindet sich im Startzustand q_s und liest das erste Zeichen. Da die ersten beiden 1en gezählt werden müssen, müssen die beiden Zeichen 0 und 1 unterschieden werden, i.e., es muß die beiden Übergänge

$$q_s \xrightarrow{0} q^u_0$$
$$q_s \xrightarrow{1} q_1$$

geben. q^u_0 ist der Zustand *ungerade Anzahl von 0en gelesen*, q_1 ist der Zustand *1 gelesen*.

Befindet sich der Automat im Zustand q^u_0 und liest nun ausschließlich 0en, wechselt er permanent zwischen den Zuständen q^u_0 und einem Zustand q^g_0 hin und her, letzterer registriert, daß eine gerade Anzahl von 0en gelesen wurde. Dies muß ein Acceptzustand sein.

Ist der Automat im Zustand q_1 und liest nun eine 0, geht er in einen Zustand q^u_{10}, über, dieser registriert, daß eine 1 und eine ungerade Anzahl von 0en gelesen wurden. Liest er im Zustand q_1 eine 1, dann muß er in einen Zustand q^g_{110} übergehen, dieser Zustand registirert, daß genau zwei 1en und eine gerade Anzahl von 0en verarbeitet wurden. Hier benutzen wir, daß keine 0 eine gerade Anzahl ist. Dieser Zustand muß ein Acceptzustand sein, da genau zwei 1en gelesen wurden.

Hat der Automat den Teilstring 10 gelesen – er befindet sich also im Zustand q^u_{10} – und liest nun wieder ausschließlich 0en, wechselt er zwischen

q^u_{10} und einem Zustand q^g_{10} hin und her. q^g_{10} ist der Zustand *eine 1 und eine gerade Anzahl von 0en gelesen*. Dies ist ein Acceptszutand. In diesen Zustand transformiert der Automat auch, wenn er eine gerade Anzahl von `0`en gelesen hat und die erste `1` wird verarbeitet.

Ist er nun in einem der beiden Zustände q^u_{10} oder q^g_{10} und liest er eine `1`, dann muß registriert werden, daß die zweite `1` gelesen wurde. Er geht daher in entsprechende Zustände q^u_{110} bzw. q^g_{110} über. Zwischen diesen Zuständen wechselt der Automat, wenn er genau zwei `1` en gelesen hat und liest jetzt nur noch `0`en. Beide sind Acceptzustände, da genau zwei `1`en gelesen wurden, unabhängig davon, wie viele `0`en. Es muß aber dennoch differenziert werden, ob eine gerade oder ungerade Anzahl von `0`en verarbeitet wurden, denn sind mehr als drei `1`en verarbeitet, muß der Automat genau dies wieder unterscheiden können.

Ist er jetzt in einem der beiden Zustände $q^{u/g}_{110}$ und liest eine `1`, tritt der Fall ein, daß mehr als zwei `1`en gelesen wurden. Dann muß der Automat in die oben definierten Zustände $q^g_{0>1}$ *gerade Anzahl von* `0`*en und mehr als zwei* `1`*en gelesen* oder $q^u_{0>1}$ *ungerade Anzahl von* `0`*en und mehr als zwei* `1`*en gelesen* übergehen, i.e. es müssen die Übergänge

$$q^u_{110} \xrightarrow{\texttt{1}} q^u_{0>1}$$
$$q^g_{110} \xrightarrow{\texttt{1}} q^g_{0>1}$$

vorliegen. Diese gehen in sich über beim Lesen einer `1` und der Automat wechselt zwischen diesen Zuständen beim Lesen von `0`. Dabei ist $q^g_{0>1}$ ein Acceptzustand.

Damit ist der Automat M_{12} formal beschreibbar durch

$$M_{12} = (Q, \Sigma, \delta, q^s, F)$$

1. Zehn Zustände: $Q = \{q_s, q^u_0, q^g_0, q_1, q^u_{10}, q^g_{10}, q^u_{110}, q^g_{110}, q^u_{0>1}, q^g_{0>1}\}$
2. Das zugrundeliegende Alphabet: $\Sigma = \{\texttt{0}, \texttt{1}\}$
3. Die Übergangsfunktion δ:

	`0`	`1`
q_s	q^u_0	q_1
q^u_0	q^g_0	q^u_{10}
q^g_0	q^u_0	q^g_{10}
q_1	q^u_{10}	q^g_{110}
q^u_{10}	q^g_{10}	q^u_{110}
q^g_{10}	q^u_{10}	q^g_{110}
q^u_{110}	q^g_{110}	$q^u_{0>1}$
q^g_{110}	q^u_{110}	$q^g_{0>1}$
$q^u_{0>1}$	$q^g_{0>1}$	$q^u_{0>1}$
$q^g_{0>1}$	$q^u_{0>1}$	$q^g_{0>1}$

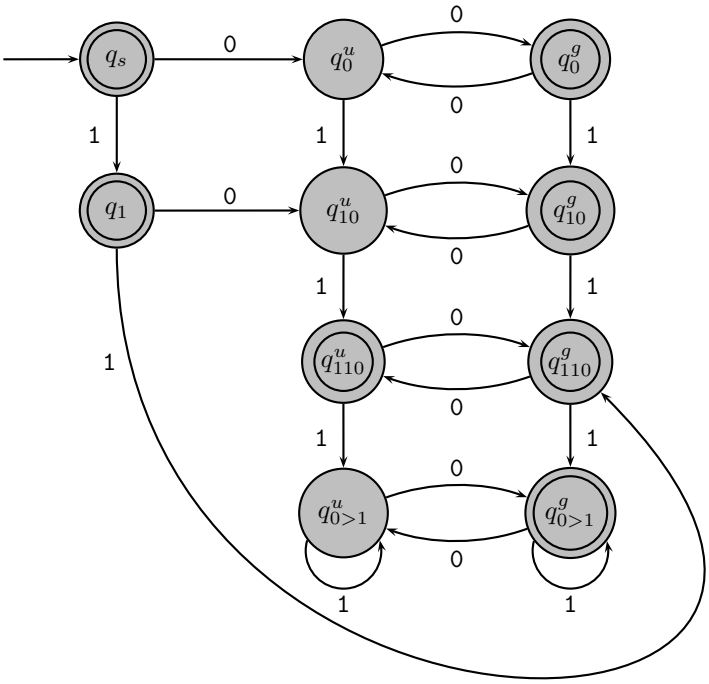


Abbildung A.18: Der deterministische endliche Automat M_{12}.

4. Der Startzustand ist q_s.
5. Die Menge der Acceptzustände:

$$F = \{q_s, q_1, q_0^g, q_{10}^g, q_{110}^u, q_{110}^g, q_{0>1}^g\}.$$

Alternative Lösungsmethode:

Die Konstruktion des Automaten M_l kann alternativ über die Produktkonstruktion durchgeführt werden. Dies folgt aus der Beobachtung, dass die Sprache

$$L = \{w \mid w \text{ enthält eine gerade Anzahl von } \texttt{0}\text{en} \\ \text{oder genau zwei } \texttt{1}\text{en}\}$$

geschrieben werden kann in der Form:

$$L_1 = \{w \mid w \text{ enthält eine gerade Anzahl von } \texttt{0}\text{en}\}$$
$$L_2 = \{w \mid w \text{ enthält genau zwei } \texttt{1}\text{en}\}$$

und L ist die mengentheoretische Vereinigung der Sprachen L_1 und L_2:

$$L = L_1 \cup L_2$$

Es gibt ein konstruktives Verfahren, wie man einen DFA für eine Sprache A konstruiert, wenn die Automaten bekannt sind, die die Sprachen L_1 und L_2 erkennen mit $L = L_1 \cup L_2$.

Die Sprache

$$L_1 = \{w \mid w \text{ enthält eine gerade Anzahl von } \texttt{0}\text{en}\}$$

wird von dem endlichen deterministischen Automaten

$$M_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$

erkannt mit:

1. der Menge der Zustände $Q_1 = \{q_g, q_s\}$
2. dem Eingabealphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$
3. dem Startzustand $q_1^s = q_g$
4. der Menge der Acceptzustände $F_1 = \{q_g\}$
5. und der Übergangsfunktion δ_1 mit

	0	1
q_g	q_u	q_g
q_u	q_g	q_u

Das Zustandsdiagramm dieses Automaten ist in der Abbildung [A.19] dargestellt.

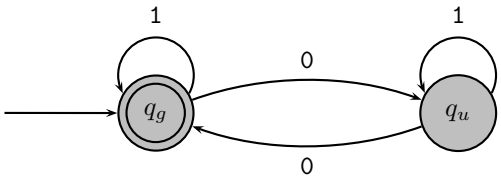


Abbildung A.19: Zustandsdiagramm des Automaten M_1.

Die Sprache

$$L_2 = \{w \mid w \text{ enthält genau zwei } \texttt{1}\text{en}\}$$

wird von dem endlichen deterministischen Automaten

$$M_2 = (Q_2, \Sigma, \delta_2, q_2^s, F_2)$$

erkannt mit:

1. der Menge der Zustände $Q_2 = \{q_0, q_1, q_{11}, q_{>1}\}$
2. dem Eingabealphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$
3. dem Startzustand $q_2^s = q_0$
4. der Menge der Acceptzustände $F_1 = \{q_{11}\}$
5. und der Übergangsfunktion δ_2 mit

	0	1
q_0	q_0	q_1
q_1	q_1	q_{11}
q_{11}	q_{11}	$q_{>1}$
$q_{>1}$	$q_{>1}$	$q_{>1}$

Das Zustandsdiagramm dieses Automaten ist in der Abbildung [A.20] dargestellt.

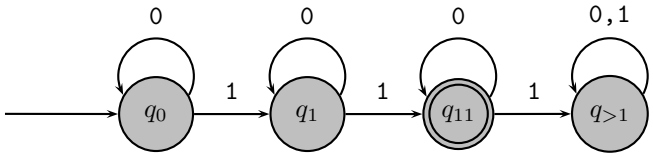


Abbildung A.20: Zustandsdiagramm des Automaten M_2.

Der Produktautomat M_{12}, der die Sprache

$$L = \{w \mid w \text{ enthält eine gerade Anzahl von } \texttt{0}\text{en} \text{ oder genau zwei } \texttt{1}\text{en}\}$$

erkennt, ergibt sich zu:

$$M_{12} = (Q, \Sigma, \delta, q^s, F)$$

mit:

1. der Menge der Zustände Q

   $$Q = Q_1 \times Q_2 = \{(r_1, r_2) \mid r_1 \in Q_1, r_2 \in Q_2\},$$

   was auf die acht Zustände führt:

   $$\begin{aligned} Q = \{&(q_g, q_0), (q_g, q_1), (q_g, q_{11}), (q_g, q_{>1}),\\ = \quad &(q_u, q_0), (q_u, q_1), (q_u, q_{11}), (q_u, q_{>1})\} \end{aligned}$$

2. dem Eingabealphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
3. dem Startzustand $q^s = (q_1^s, q_2^s) = (q_g, q_0)$
4. der Menge der Acceptzustände $F = F_1 \times Q_2 \cup Q_1 \times F_2$, i.e.

$$\begin{aligned} F &= \{q_g\} \times \{q_0, q_1, q_{11}, q_{>1}\} \cup \{q_g, q_u\} \times \{q_{11}\} \\ &= \{(q_g, q_0), (q_g, q_1), (q_g, q_{11}), (q_g, q_{>1}), (q_u, q_{11})\} \end{aligned}$$

5. und der Übergangsfunktion δ mit

$$\delta((r_1, r_2), a) = \big(\delta_1(r_1, a), \delta_2(r_2, a)\big)$$

für jedes $(r_1, r_2) \in Q$ und $a \in \Sigma$.

	0	1
(q_g, q_0)	(q_u, q_0)	(q_g, q_1)
(q_g, q_1)	(q_u, q_1)	(q_g, q_{11})
(q_g, q_{11})	(q_u, q_{11})	$(q_g, q_{>1})$
$(q_g, q_{>1})$	$(q_u, q_{>1})$	$(q_g, q_{>1})$
(q_u, q_0)	(q_g, q_0)	(q_u, q_1)
(q_u, q_1)	(q_g, q_1)	(q_u, q_{11})
(q_u, q_{11})	(q_g, q_{11})	$(q_u, q_{>1})$
$(q_u, q_{>1})$	$(q_g, q_{>1})$	$(q_u, q_{>1})$

Diese Übergangsfunktion führt auf das Zustandsdiagramm [A.21]:

Zu 13 In diesem Beispiel ist ein deterministischer endlicher Automat zu konstruieren, der die leere Menge akzeptiert. Die reguläre Sprache, die der leeren Menge $\emptyset$ entspricht, ist diejenige, deren sämtliche Strings nicht akzeptiert werden. Also, es ist ein Automat zu konstruieren, der keine Strings akzeptiert. Dies kann einfach durch die in Abbildung [A.22] dargestellt Maschine realisiert werden.

Zu 14 Es ist ein Automat zu konstruieren, der sämtliche Strings mit Ausnahme des leeren Strings λ akzeptiert.

Der Automat befindet sich in einem Startzustand q_s und liest einen String. Unabhängig davon, welches Zeichen gelesn wird, geht er in einen Zustand q_a über, der Acceptzustand ist. In diesem Zustand bleibt er, egal, welche und wieviele Zeichen gelesen werden. q_a hat also in sich übergehende `0,1` Zustände. Der Startzustand darf kein Acceptzustand sein, da sich der Automat in q_s befindet, wenn er noch kein Zeichen — *i.e.* den leeren String — gelesen hat. Das Zustandsdiagramm dieses Automaten ist in Abbildung [A.23] dargestellt.

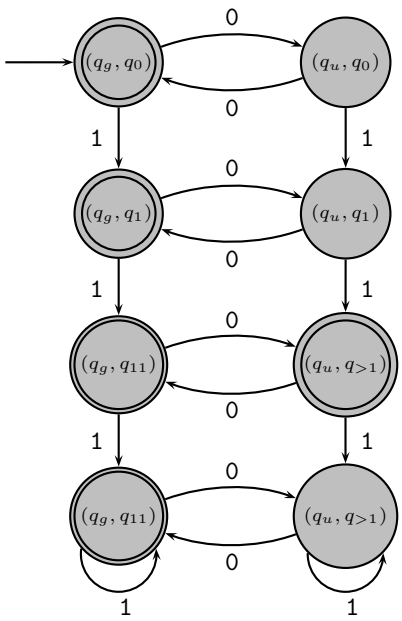


Abbildung A.21: Zustandsdiagramm des Produktautomaten M_l.

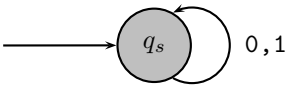


Abbildung A.22: Der Automat, der die Sprache $\emptyset$ akzeptiert.

A.3.13 Übung [3.33]

Geben Sie reguläre Ausdrücke an, die die folgenden Sprachen generieren:

1. $\{w \mid w$ beginnt mit einer `1` und endet mit einer `0` $\}$.
2. $\{w \mid w$ enthält mindestens drei `1`en $\}$.
3. $\{w \mid w$ enthält den Substring `0101`, i.e. $w = x$`0101`y für x, y $\}$.
4. $\{w \mid w$ hat mindestens die Länge drei und enthält als drittes Symbol die `0` $\}$.
5. $\{w \mid w$ beginnt mit einer `0` und hat ungerade Länge oder beginnt mit einer `1` und hat gerade Länge $\}$.

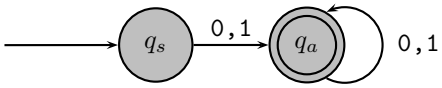


Abbildung A.23: Der Automat M_{14}, der alle Strings mit Ausnahme von λ akzeptiert.

6. $\{w \mid w$ enthält nicht den Substring `110` $\}$.

7. $\{w \mid$ die Länge von w ist höchstens 5 $\}$.

8. $\{w \mid w$ ist jeder beliebige String mit Ausnahme von `11` und `111` $\}$.

9. $\{w \mid$ jede ungerade Position von w ist eine `1` $\}$.

10. $\{w \mid w$ enthält wenigstens zwei `0`en und höchstens eine `1` $\}$.

11. $\{\lambda, 0\}$.

12. $\{w \mid w$ enthält eine gerade Anzahl von `0`en oder genau zwei `1`en $\}$.

13. Die leere Menge.

14. Alle Strings mit Ausnahme des leeren Strings.

Lösungen:

In den folgenden Lösungen betrachten wir das Alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. Dann ist Σ^* die Menge aller Strings über Σ. Diese Sprache enthält den leeren String λ, alle Worte der Länge 1, der Länge 2 usw. über dem Alphabet $\{\mathtt{0}, \mathtt{1}\}$.

Zu 1 Der reguläre Ausdruck, der die Sprache

$$L = \{w \mid w \text{ beginnt mit einer } \mathtt{1} \text{ und endet mit einer } \mathtt{0} \}$$

generiert, ist

$$R = \mathtt{1}\Sigma^*\mathtt{0}$$

Begründung:

$$
\begin{aligned}
R &= \mathtt{1}\Sigma^*\mathtt{0} \\
&= \{\mathtt{1}\} \circ \big(\{\mathtt{0}\} \cup \{\mathtt{1}\}\big)^* \circ \{\mathtt{0}\} \\
&= \{\mathtt{1}\} \circ \{\mathtt{0},\mathtt{1}\}^* \circ \{\mathtt{0}\} \\
&= \{\mathtt{1}\} \circ \Big(\{\lambda\} \cup \{\mathtt{0},\mathtt{1}\} \cup \{\mathtt{00},\mathtt{01},\mathtt{10},\mathtt{11}\} \cup \{\mathtt{000},\ldots\} \cup \ldots\Big)^* \circ \{\mathtt{0}\} \\
&= \{\mathtt{1}\} \circ \Big(\{\lambda,\mathtt{0},\mathtt{1},\mathtt{00},\mathtt{01},\mathtt{10},\mathtt{11},\mathtt{000},\ldots\}\Big) \circ \{\mathtt{0}\} \\
&= \Big(\{\mathtt{1},\mathtt{10},\mathtt{11},\mathtt{100},\mathtt{101},\mathtt{110},\mathtt{111},\mathtt{1000},\ldots\}\Big) \circ \{\mathtt{0}\} \\
&= \big\{\mathtt{10},\mathtt{100},\mathtt{110},\mathtt{1000},\mathtt{1010},\mathtt{1100},\mathtt{1110},\mathtt{10000},\ldots\big\}.
\end{aligned}
$$

Bei diesen Manipulationen werden mehrfach die Mengenoperationen (*e.g.* Konkatenation) benutzt.

Zu 2 Der reguläre Ausdruck, der die Sprache

$$L = \{w \mid w \text{ enthält mindestens drei } \mathtt{1}\text{en }\}$$

generiert, ist durch

$$R = \Sigma^*\mathtt{1}\Sigma^*\mathtt{1}\Sigma^*\mathtt{1}\Sigma^*$$

gegeben.

Zu 3 Der reguläre Ausdruck, der die Sprache

$$L = \{w \mid w \text{ enthält den Substring } \mathtt{0101}, \textit{ i.e. } w = x\mathtt{0101}y \text{ für } x,\ y\ \}$$

generiert, ist

$$R = \Sigma^*\mathtt{0101}\Sigma^*$$

Zu 4 Der reguläre Ausdruck, der die Sprache

$$L = \{w \mid w \text{ hat mindestens die Länge drei und enthält als drittes Symbol die } \mathtt{0}\}$$

generiert, ist

$$R = \Sigma\,\Sigma\,\mathtt{0}\,\Sigma^*$$

Begründung:

Dieser reguläre Ausdruck ist eine Kurzform für

$$
\begin{aligned}
R &= \Sigma\,\Sigma\,\mathtt{0}\,\Sigma^* \\
&= (\{\mathtt{0}\} \cup \{\mathtt{1}\}) \circ (\{\mathtt{0}\} \cup \{\mathtt{1}\}) \circ \{\mathtt{0}\} \circ (\{\mathtt{0}\} \cup \{\mathtt{1}\})^* \\
&= (\{\mathtt{0},\mathtt{1}\}) \circ (\{\mathtt{0},\mathtt{1}\}) \circ \{\mathtt{0}\} \circ (\{\mathtt{0}\} \cup \{\mathtt{1}\})^* \\
&= \{\mathtt{00},\mathtt{01},\mathtt{10},\mathtt{11}\}) \circ \{\mathtt{0}\} \circ (\{\mathtt{0}\} \cup \{\mathtt{1}\})^* \\
&= \{\mathtt{000},\mathtt{010},\mathtt{100},\mathtt{110}\}) \circ \{\mathtt{0},\mathtt{1}\}^* \\
&= \{\mathtt{000},\mathtt{010},\mathtt{100},\mathtt{110}\}) \circ \{\epsilon,\mathtt{0},\mathtt{1},\mathtt{00},\mathtt{01},\mathtt{10},\mathtt{11},\mathtt{000},\mathtt{001},\mathtt{010},\ldots\} \\
&= \{\mathtt{000},\mathtt{010},\mathtt{100},\mathtt{110},\mathtt{0000},\mathtt{0100},\mathtt{1000},\mathtt{1100},\mathtt{0001},\mathtt{0101},\mathtt{1001},\mathtt{1101},\ldots\}
\end{aligned}
$$

Zu 5 Die Sprache

$$L = \{w \mid w \text{ beginnt mit einer } \mathtt{0} \text{ und hat ungerade Länge}$$
$$\text{oder beginnt mit einer } \mathtt{1} \text{ und hat gerade Länge}\}$$

wird durch den regulären Ausdruck

$$R = \mathtt{0}(\Sigma\Sigma)^* \ \cup \ \mathtt{1}\big((\Sigma\Sigma)^*\Sigma\big)$$

generiert. Begründung:

- Der Teilausdruck $\mathtt{0}(\Sigma\Sigma)^*$ beschreibt Strings, die mit dem Symbol $\mathtt{0}$ beginnen. Es folgen stets eine gerade Anzahl von $\mathtt{0}$en bzw. $\mathtt{1}$en, so daß dies also Strings sind mit ungerader Länge, die mit $\mathtt{0}$ beginnen. Da Σ als Menge der Strings der Länge 1 über dem Alphabet Σ interpretiert werden kann, ist $\Sigma \circ \Sigma$ die Menge der Strings über Σ der Länge 2, i.e.:
$$\Sigma \circ \Sigma = \{\mathtt{0},\mathtt{1}\} \circ \{\mathtt{0},\mathtt{1}\} = \{\mathtt{00},\mathtt{01},\mathtt{10},\mathtt{11}\}$$
Damit ist
$$\begin{aligned}(\Sigma \circ \Sigma)^* &= (\Sigma \circ \Sigma)^0 \cup (\Sigma \circ \Sigma)^1 \circ (\Sigma \circ \Sigma)^2 \ldots \\ &= \big(\{\mathtt{0},\mathtt{1}\} \circ \{\mathtt{0},\mathtt{1}\}\big)^* \\ &= \{\mathtt{00},\mathtt{01},\mathtt{10},\mathtt{11}\}^* \\ &= \{\epsilon\} \cup \{\mathtt{00},\mathtt{01},\mathtt{10},\mathtt{11}\} \cup \\ &\qquad \{\mathtt{0000},\mathtt{0001},\mathtt{0010},\mathtt{0011},\ldots,\mathtt{1111}\} \cup \ldots\end{aligned}$$
Daher beschreibt der Teilausdruck $\mathtt{0}(\Sigma\Sigma)^*$ Strings über dem Alphabet $\Sigma = \{\mathtt{0},\mathtt{1}\}$, die mit dem Symbol $\mathtt{0}$ beginnen und ungerade Länge haben.
- Analog konstruiert man den zweiten Anteil. Da
$$\Sigma \circ \Sigma = \{\mathtt{0},\mathtt{1}\} \circ \{\mathtt{0},\mathtt{1}\} = \{\mathtt{00},\mathtt{01},\mathtt{10},\mathtt{11}\}$$
Strings der Länge 2 generiert, sind — wie oben gezeigt — $(\Sigma \circ \Sigma)^*$ Strings gerader Länge über $\{\mathtt{0},\mathtt{1}\}$. Damit ist
$$\begin{aligned}(\Sigma \circ \Sigma)^* \circ \Sigma &= \Big((\Sigma \circ \Sigma)^0 \cup (\Sigma \circ \Sigma)^1 \circ (\Sigma \circ \Sigma)^2 \ldots\Big) \circ \Sigma \\ &= \Big(\{\mathtt{0},\mathtt{1}\} \circ \{\mathtt{0},\mathtt{1}\}\Big)^* \circ \{\mathtt{0},\mathtt{1}\} \\ &= \Big(\{\mathtt{00},\mathtt{01},\mathtt{10},\mathtt{11}\}^*\Big) \circ \{\mathtt{0},\mathtt{1}\} \\ &= \Big(\{\epsilon\} \cup \{\mathtt{00},\mathtt{01},\mathtt{10},\mathtt{11}\} \cup \\ &\qquad \{\mathtt{0000},\mathtt{0001},\mathtt{0010},\mathtt{0011},\ldots,\mathtt{1111}\} \cup \ldots\Big) \circ \{\mathtt{0},\mathtt{1}\} \\ &= \{\mathtt{0},\mathtt{1}\} \cup \{\mathtt{000},\mathtt{010},\mathtt{100},\mathtt{110},\mathtt{001},\mathtt{011},\mathtt{101},\mathtt{111}\} \cup \\ &\qquad \{\mathtt{00000},\mathtt{00010},\ldots\} \cup \ldots\end{aligned}$$

Wird dieser Ausdruck noch mit der 1 konkateniert, folgt, dass

$$1\big((\Sigma\Sigma)^*\Sigma\big)$$

Strings gerader Länge über dem Alphabet $\Sigma = \{0, 1\}$ erzeugt.

- Die Vereinigung dieser beiden Teilausdrücke erzeugt die geforderte Sprache.

Zu 6 Die Sprache

$$L = \{w \mid w \text{ enthält nicht den Substring } 110\ \}$$

wird generiert durch den regulären Ausdruck

$$R = 0^*(\Sigma 0)^* 0^* 1^*$$

Begründung: Wir entwickeln den regulären Ausdruck bis zur Länge von vier Zeichen und überzeugen uns, dass darin der Substring 110 nicht vorkommt.

$$\begin{aligned}
R &= 0^*(\Sigma 0)^* 0^* 1^* \\
&= \{0\}^*\big(\{0,1\} \circ \{0\}\big)^* \circ \{0\}^* \circ \{1\}^* \\
&= \{\lambda, 0, 00, 000, 0000, \ldots\}\circ \\
&\quad \circ \big(\{00, 10\}\big)^*\circ \\
&\quad \circ \{\epsilon, 0, 00, 000, 0000, \ldots\}\circ \\
&\quad \circ \{\epsilon, 1, 11, 111, 1111, \ldots\} \\
&= \{\lambda, 0, 00, 000, 0000, \ldots\}\circ \\
&\quad \circ \{\epsilon, 00, 10, 0000, 0010, 1000, 1010, \ldots\}\circ \\
&\quad \circ \{\epsilon, 0, 00, 000, 0000, \ldots\}\circ \\
&\quad \circ \{\epsilon, 1, 11, 111, 1111, \ldots\} \\
&= \{\lambda, 0, 00, 000, 0000, \ldots\}\circ \\
&\quad \circ \{\epsilon, 00, 10, 0000, 0010, 1000, 1010, \ldots\}\circ \\
&\quad \circ \{\epsilon, 0, 1, 00, 11, 01, 000, 111, 011, 001, \\
&\qquad 0000, 1111, 0111, 1000, 0011, \ldots\} \\
&= \{\lambda, 0, 00, 10, 000, 010, 0000, 0010, 1000, 1010, \ldots\}\circ \\
&\quad \circ \{\epsilon, 0, 1, 00, 11, 000, 111, 011, 001, \\
&\qquad 0000, 1111, 0111, 1000, 0011, 0001 \ldots\} \\
&= \Big\{\epsilon, 0, 1, 00, 01, 10, 11, \\
&\qquad 000, 001, 010, 011, 100, 101, 111, \\
&\qquad 0000, 0001, 0010, 0011, 0100, 0101, 0111, \\
&\qquad 1000, 1001, 1010, 1011, 1111, \ldots\Big\}
\end{aligned}$$

Eine systematische Herleitung des regulären Ausdrucks der L generiert aus einem DFA ist Thema der Übung [3.39].

Zu 9 Die Sprache

$$L = \{w \mid \text{ jede ungerade Position von } w \text{ ist eine } \mathtt{1} \}$$

wird generiert durch den regulären Ausdruck

$$R = (\mathtt{1}\Sigma)^* \cup (\mathtt{1}\Sigma)^*\mathtt{1}.$$

Begründung:

$$\begin{aligned}
(\mathtt{1}\Sigma)^* &= \big(\{\mathtt{1}\} \circ \Sigma\big)^* \\
&= \big(\{\mathtt{1}\} \circ \{\mathtt{0},\mathtt{1}\}\big)^* \\
&= \big(\{\mathtt{10},\mathtt{11}\}\big)^* \\
&= (\{\lambda\} \cup \{\mathtt{10},\mathtt{11}\} \cup \{\mathtt{1010},\mathtt{1011},\mathtt{1110},\mathtt{1111}\} \cup \\
&\quad \{\mathtt{101010},\mathtt{101011},\mathtt{101110},\mathtt{101111}, \\
&\quad \mathtt{111010},\mathtt{111011},\mathtt{111110},\mathtt{111111}\} \cup \ldots \\
&= \{\lambda,\mathtt{10},\mathtt{11},\mathtt{1010},\mathtt{1011},\mathtt{1110},\mathtt{1111},\mathtt{101010},\mathtt{101011},\mathtt{101110},\mathtt{101111}, \\
&\quad \mathtt{111010},\mathtt{111011},\mathtt{111110},\mathtt{111111},\ldots\}
\end{aligned}$$

Dieser Anteil generiert alle Bitstrings, an deren ungerader Position eine 1 steht, mit gerader Länge. Die Bitstrings mit dieser Eigenschaft mit ungerader Länge werden durch den zweiter Term generiert.

Zu 10 Die reguläre Sprache

$$L = \{w \mid w\text{enthält wenigstens zwei } \mathtt{0}\text{en und höchstens eine } \mathtt{1} \}$$

wird durch den folgenden reguläre Ausdruck generiert:

Zu 11 $\{\lambda, 0\}$.

Zu 12 $\{w \mid w$ enthält eine gerade Anzahl von 0en oder genau zwei 1en $\}$.

Zu 13 Die leere Menge.

Zu 14 Der reguläre Ausdruck

$$R = (\mathtt{0} \cup \mathtt{1})\Sigma^* = \Sigma \circ (\Sigma)^*$$

generiert alle Strings mit Ausnahme des leeren Strings.

A.3.14 Übung [3.34]

Ein Spiel ist definiert durch das zweimalige Werfen mit einem Würfel mit den Zahlen 1 bis 6. Das Spiel gilt als gewonnen, wenn die Summe der Augenzahlen durch 3 teilbar ist. Im Folgenden Sei der Ablauf eines Spiels codiert als ein Wort

$$W \in \{1,2,3,4,5,6\}^2,$$

wobei das erste Zeichen für den ersten Wurf und das zweite für den zweiten Wurf steht.

(a) Entwickeln Sie eine `DFA` M, welcher die Kodierung eines Spielablaufs genau dann akzeptiert, wenn das Spiel als gewonnen gilt. Geben Sie den Automaten vollständig an.

(b) Minimieren Sie den Automaten M und geben Sie den minimierten `DFA` M' vollständig an.

(c) Verallgemeinern Sie den endlichen Automaten M bzw M', so dass er bei einem beliebig langen Würfelspiel mit einem Würfel genau dann akzeptiert, wenn die Summe der Augenzahlen aller Würfe durch 3 teilbar ist.

Lösung:

(a) Der `DFA`, der den Spielablauf kodiert, *i.e.* der genau dann akzeptiert, wenn die Augensumme nach zwei Würfen durch 3 teilbar ist. Mit anderen Worten, wenn in zwei Würfen die Augenzahlen

   (a) 1,2 oder 1,5
   (b) 2,1 oder 2,4
   (c) 3,3 oder 3,6
   (d) 4,2 oder 4,5
   (e) 5,1 oder 5,4
   (f) 6,3 oder 6,6

   fallen ist zu akzeptieren, anderfalls wird verworfen. Der `DFA`, der diese Sprache akzeptiert ist das Fünftupel

$$M = (Q, \Sigma, \delta, q^s, F)$$

   mit den Zuständen

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\},$$

   dem Eingabealphabet

$$\Sigma = \{1,2,3,4,5,6\},$$

   dem Startzustand $q^s = q_0$, dem Acceptzustand q_7 und der Übergangsfunktion δ:

	1	2	3	4	5	6
q_0	q_1	q_2	q_3	q_4	q_5	q_6
q_1	q_8	q_7	q_8	q_8	q_7	q_8
q_2	q_7	q_8	q_8	q_7	q_8	q_8
q_3	q_8	q_8	q_7	q_8	q_8	q_7
q_4	q_8	q_7	q_8	q_8	q_7	q_8
q_5	q_7	q_8	q_8	q_7	q_8	q_8
q_6	q_8	q_8	q_7	q_8	q_8	q_7
q_7	q_8	q_8	q_8	q_8	q_8	q_8
q_8	q_8	q_8	q_8	q_8	q_8	q_8

Das Zustandsdiagramm dieses Automaten ist in der Abbildung [A.24] dargestellt.

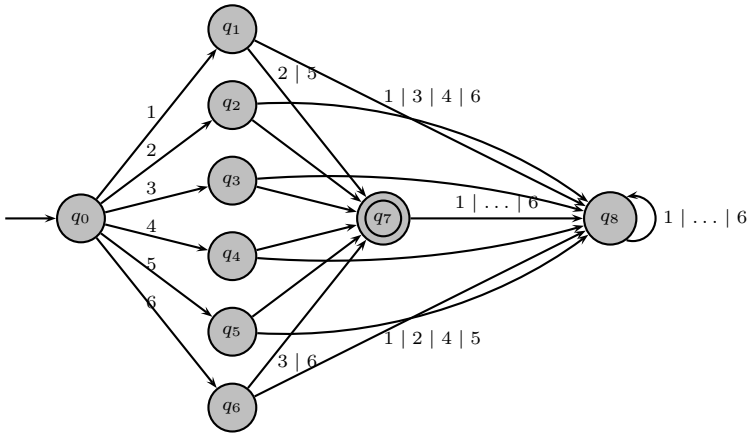


Abbildung A.24: Zustandsdiagramm des Spielautomaten.

Der Übersichtlichkeit halber sind in dem Diagramm [A.24] nicht alle Bezeichnungen der Übergänge aufgeführt. Die Systematik sollte klar und anhand der angegebenen Beschriftungen ersichtlich sein. Die Folge von Übergängen

$$q_0 \xrightarrow{1} q_1 \xrightarrow{2|5} q_7$$

führt den Startzustand in den Acceptzustand über, die Summe der beiden Pfeilbeschriftungen ergibt eine durch 3 teilbare Zahl. Hingegen führt die

Folge von Übergängen

$$q_0 \xrightarrow{1} q_1 \xrightarrow{1|3|4|6} q_8$$

auf eine Augenzahl, die nicht durch 3 teilbar ist, der Automat akzeptiert dieses Wort nicht. Die anderen Übergänge ergeben sich in analoger Weise.

(b) Wir sehen uns die Übergänge des `DFA` in der Abbildung [A.24], die in den Acceptzustand übergehen, im Detail an:

$$\begin{aligned}
q_0 &\xrightarrow{1} q_1 \xrightarrow{2|5} q_7,\\
q_0 &\xrightarrow{2} q_2 \xrightarrow{1|4} q_7,\\
q_0 &\xrightarrow{3} q_3 \xrightarrow{3|6} q_7,\\
q_0 &\xrightarrow{4} q_4 \xrightarrow{2|5} q_7,\\
q_0 &\xrightarrow{5} q_5 \xrightarrow{1|4} q_7,\\
q_0 &\xrightarrow{6} q_6 \xrightarrow{3|6} q_7.
\end{aligned}$$

An dieser Aufstelleung erkennt man, dass die Übergänge

$$\begin{aligned}
q_0 &\xrightarrow{1} q_1 \xrightarrow{2|5} q_7,\\
q_0 &\xrightarrow{4} q_4 \xrightarrow{2|5} q_7,
\end{aligned}$$

sowie

$$\begin{aligned}
q_0 &\xrightarrow{2} q_2 \xrightarrow{1|4} q_7,\\
q_0 &\xrightarrow{5} q_5 \xrightarrow{1|4} q_7,
\end{aligned}$$

und

$$\begin{aligned}
q_0 &\xrightarrow{3} q_3 \xrightarrow{3|6} q_7,\\
q_0 &\xrightarrow{6} q_6 \xrightarrow{3|6} q_7
\end{aligned}$$

gleichwertig sind, *i.e.* die gleichen Übergänge aufweisen. Daher können die Zustände q_1 und q_4, q_2 und q_5 sowie q_3 und q_6 zusammengezogen werden. Wie benötigen daher nicht den Minimalisierungsalgorithmus um den Minimalautomaten zu erhalten.

Das Zustandsdiagramm des Minimalautomaten ist in der Abbildung [A.25] dargestellt.

(c) Der allgemeine Automat ist einfacher als die beiden Automaten zuvor. Wir müssen in den Zuständen lediglich die Reste speichern, *i.e.*

$$\begin{aligned}
q_0 &\longleftrightarrow \text{Rest 0 bei den Augenzahlen}\\
q_1 &\longleftrightarrow \text{Rest 1 bei den Augenzahlen}\\
q_2 &\longleftrightarrow \text{Rest 2 bei den Augenzahlen}
\end{aligned}$$

Damit ist q_0 Startzustand und Acceptzustand. Die Übergangsfunktion δ wird

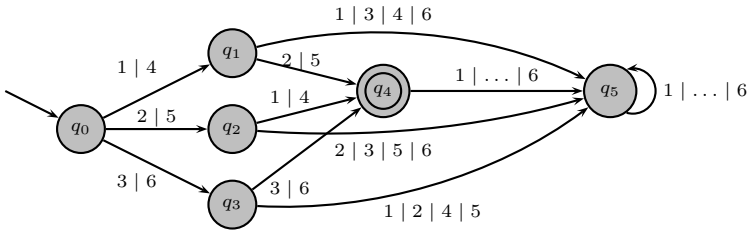
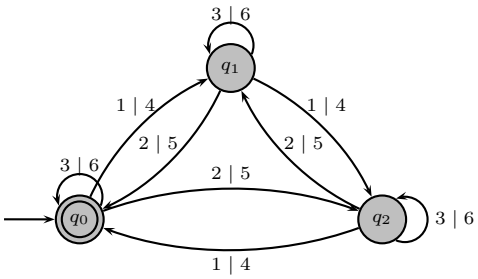


Abbildung A.25: Zustandsdiagramm des Minimalautomaten.

	1	2	3	4	5	6
q_0	q_1	q_2	q_0	q_1	q_2	q_0
q_1	q_2	q_0	q_1	q_2	q_0	q_1
q_2	q_0	q_1	q_2	q_0	q_1	q_2



A.3.15 Übung [3.35]

Gegeben ist der reguläre Audruck

$$R = \texttt{01}(\texttt{0} + \texttt{1})^*\texttt{10}^*.$$

Entwickeln Sie einen nichtdeterministischen endlichen Automaten N und formen Sie diesen anschließend in einen deterministischen endlichen Automaten M um mit $L(R) = L(M) = L(N)$.

1. Verwenden Sie dazu das Potenzmengenverfahren.
2. Verwenden Sie das Verfahren über den Zustandsbaum.

Lösung:

Die Sprache, die der reguläre Ausdruck R generiert, sind die Bitstrings, die mit `01` beginnen und mindesten eine weitere `1` enthaltewn. Zwischen dem Präfix `01` können beliebige Teilstrings aus `0`en und `1`en (oder der leere String) liegen, hinter der letzten `1` können beliebig viele `0`en folgen.

Ein `NFA`, der diese Spracher erkennt ist durch folgendes 5–Tupel gegeben:

$$N = (Q, \Sigma, \delta, q^s, F)$$

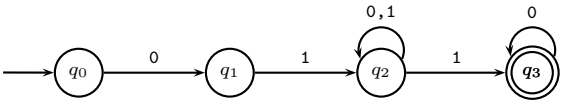
mit

$$Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{\texttt{0}, \texttt{1}\}, q^2 = q_0, F = \{q_3\}$$

und der Übergangsrelation δ:

	`0`	`1`
q_0	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2, q_3\}$
q_3	$\{q_3\}$	$\emptyset$

Das Zustandsdiagramm ist:



Zur Umwandlung des `NFA` verwenden wir die Potenzmengenkonstruktion aus dem Kapitel [3.6]. Der `DFA`, der die gleiche Sprache akzeptiert, ist durch das folgende 5–Tupel gegeben:

$$M = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$

mit

$$\begin{aligned}
Q_1 &= \wp(Q) \\
&= \wp(\{q_0, q_1, q_2, q_3\}) \\
&= \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_3\}, \\
&\quad \{q_0, q_1\}, \{q_0, q_2\}, \{q_0, q_3\}, \{q_1, q_2\}, \{q_1, q_3\}, \{q_2, q_3\} \\
&\quad \{q_0, q_1, q_2\}, \{q_0, q_1, q_3\}, \{q_0, q_2, q_3\}, \{q_1, q_2, q_3\}, \\
&\quad \{q_0, q_1, q_2, q_3\}\}
\end{aligned}$$

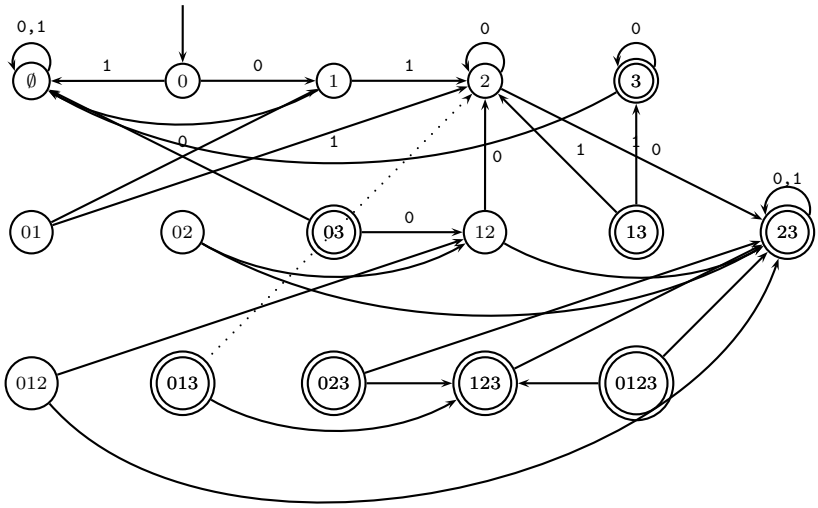
Dieser DFA hat somit $2^4 = 16$ Zustände. Der Startzustand ist $q_1^s = \{q_0\}$, die Menge der Acceptzustände besteht aus allen Teilmengen, die einen Zustand q_3 enthalten:

$$\begin{aligned}
F_1 = \{&\{q_3\}, \{q_0, q_3\}, \{q_1, q_3\}, \{q_2, q_3\}, \{q_0, q_1, q_3\}, \\
&\{q_0, q_2, q_3\}, \{q_1, q_2, q_3\}, \{q_0, q_1, q_2, q_3\}\}.
\end{aligned}$$

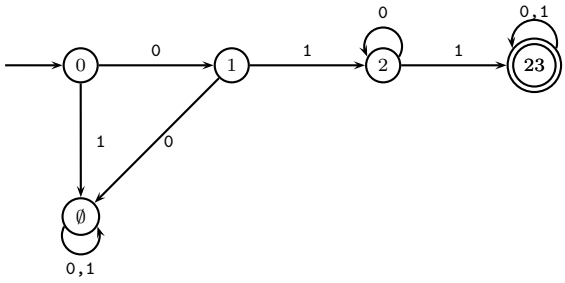
Die Übergangsfunktion ist:

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_1\}$	$\emptyset$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$\{q_2\}$	$\{q_2\}$	$\{q_2, q_3\}$
$\{q_3\}$	$\{q_3\}$	$\emptyset$
$\{q_0, q_1\}$	$\{q_1\}$	$\{q_2\}$
$\{q_0, q_2\}$	$\{q_1, q_2\}$	$\{q_2, q_3\}$
$\{q_0, q_3\}$	$\{q_1, q_2\}$	$\emptyset$
$\{q_1, q_2\}$	$\{q_2\}$	$\{q_2, q_3\}$
$\{q_1, q_3\}$	$\{q_3\}$	$\{q_2\}$
$\{q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\}$
$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2, q_3\}$
$\{q_0, q_1, q_3\}$	$\{q_1, q_2, q_3\}$	$\{q_2\}$
$\{q_0, q_2, q_3\}$	$\{q_1, q_2, q_3\}$	$\{q_2, q_3\}$
$\{q_1, q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\}$
$\{q_0, q_1, q_2, q_3\}$	$\{q_1, q_2, q_3\}$	$\{q_2, q_3\}$

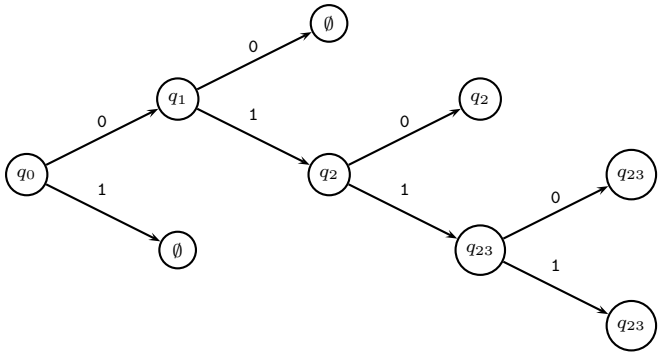
Das vollständige Zustandsdiagramm dieses DFA ist (wir nennen der Übersicht halber die Zustände hier 1 für $\{q_1\}$ usw.; die Übergänge sind nicht vollständig benannt):



Diese Zustandsdiagramm zeigt ein typisches Phänomen der Potenzmengenkonstruktion. Viele der Zustände können gestrichen werden, ohne dass sich die Sprache ändert, die der DFA akzeptiert. Beispielweise sind die Zustände $01, 02, 03$ usw. nicht vom Startzustand $\{q_0\}$ aus erreichbar. Daher können solche Zustände unberücksichtigt bleiben. Es gibt für DFA ein systematisches Verfahren, den Minimalautomaten zu erhalten.



Das Verfahren der Konstruktion des DFA über den Zustandsbaum ist:



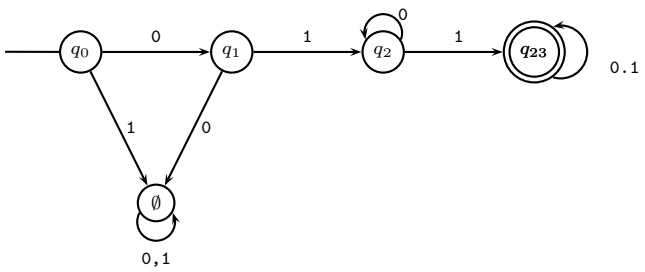
Hieraus läßt sich unmittelbar der DFA konstruieren. Wir benötigen fünf Zustände

$$Q = \{\{q_0\}, \emptyset, \{q_1\}, \{q_2\}, \{q_2, q_3\}\} = \{q_0, \emptyset, q_1, q_2, q_{23}\},$$

der Startzustand ist $\{q_0\}$, Acceptzustände: $F = \{\{q_2, q_3\}\} = \{q_{23}\}$, Die Übergangsfunktion δ ist:

	0	1
q_0	q_1	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	q_2
q_2	q_2	q_{23}
q_{23}	q_{23}	q_{23}

Das Zustandsdiagramm dieses DFA ist:



A.3.16 Übung [3.36]

Betrachten Sie die regulären Ausdrücke

$$\alpha_1 = \mathtt{01}$$
$$\alpha_2 = \mathtt{0} + \mathtt{11}$$
$$\alpha_3 = (\mathtt{01})^*.$$

Konstruieren Sie zu jedem dieser regulären Ausdrücke einen NFA nach den Konstruktionsverfahren aus dem Kapitel [3.6].

Lösung:

(a) Der reguläre Ausdruck $\alpha_1 = \mathtt{01}$ entsteht durch die Konkatenationsoperation

$$\mathtt{0} \circ \mathtt{1}.$$

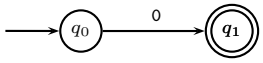
Der NFA, der die Sprache $L_1 = \{\mathtt{0}\}$ erkennt ist

$$N_1 = (\{q_0, q_1\}, \{\mathtt{0}, \mathtt{1}\}, \delta_1, q_0, \{q_1\})$$

und die Übergangsrelation δ_1 ist

	0	1
q_0	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\emptyset$

Das Zustandsdiagrammm ist



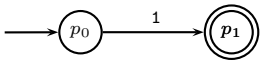
In analoger Weise ergibt sich ein NFA N_2 für die Sprache $L_2 = \{\mathtt{1}\}$. Der NFA, der 1 erkennt ist

$$N_2 = (\{p_0, p_1\}, \{\mathtt{0}, \mathtt{1}\}, \delta_2, p_0, \{p_1\})$$

und die Übergangsrelation δ_2 ist

	0	1
p_0	$\emptyset$	$\{p_1\}$
p_1	$\emptyset$	$\emptyset$

Das Zustandsdiagrammm ist



Den Produktautomaten N, der die Sprache $\{\texttt{01}\}$ akzeptiert, erhält man wie folgt:

$$N = (Q, \{\texttt{0}, \texttt{1}\}, \delta, q^s, F),$$

mit der Zustandsmenge

$$Q = Q_1 \cup Q_2 = \{q_0, q_1, p_0, p_1\}.$$

Der Startzustand des Produktautomaten ist der Startzustand des Teilautomaten N_1, *i.e.*

$$q^s = q_0.$$

Die Menge der Acceptzustaände des Produktautomaten ist F_2, *i.e.*

$$F = F_2 = \{p_1\}.$$

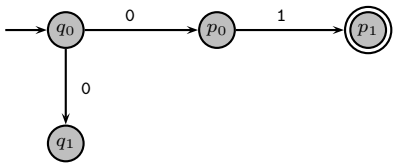
Die Übergangsrelation ist gemäß Konstruktion:[3]

$$\begin{aligned}\delta &= \delta_1 \cup \delta_2 \cup \{(p, a, p_0) \mid (p, a, q) \in \delta_1, q \in F_1\} \\ &= \delta_1 \cup \delta_2 \cup \{(q_0, \texttt{0}, p_0)\}.\end{aligned}$$

In tabellarischer Form lautet die Übergangsrelation:

	0	1
q_0	$\{q_1, p_0\}$	$\emptyset$
q_1	$\emptyset$	$\emptyset$
p_0	$\emptyset$	$\{p_1\}$
q_1	$\emptyset$	$\emptyset$

Das Zustandsdiagramm dieses Automaten ist:



[3] Wir haben den Fall hier vorliegen, dass der Startzustand des ersten Automaten kein Acceptzustand ist.

Dieser Produktautomat kann noch minimiert werden ohne dass sich die Sprache ändert, die der Automat akzeptiert. So kann der Zustand q_1 einfach wegfallen.

(b) Die Sprache, die der reguläre Ausdruck

$$\alpha = \mathtt{0} + \mathtt{11}$$

generiert, ist

$$L = L(\alpha) = \{\mathtt{0}, \mathtt{11}\},$$

i.e. die Menge der Wörter, die aus `0` *oder* `11` bestehen. Dies können wir aber schreiben als Vereingung:

$$L = L_1 \cup L_2 = \{\mathtt{0}\} \cup \{\mathtt{11}\},$$

mit den beiden Sprachen

$$L_1 = \{\mathtt{0}\} \text{ und } L_2 = \{\mathtt{11}\}$$

Das Ziel ist nun, aus den beiden `NFA`, die L_1 bzw. L_2 erzeugen, einen Produktautomaten zu konstruieren, der die Sprache $L = L_1 \cup L_2$ akzeptiert.

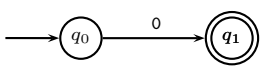
Der `NFA`, der die Sprache $L_1 = \{\mathtt{0}\}$ erkennt ist

$$N_1 = (\{q_0, q_1\}, \{\mathtt{0}, \mathtt{1}\}, \delta_1, q_0, \{q_1\})$$

und die Übergangsrelation δ_1 ist

	`0`	`1`
q_0	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\emptyset$

Das Zustandsdiagrammm ist



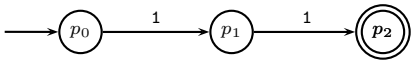
Der `NFA`, der die Sprache $L_2 = \{\mathtt{11}\}$ akzeptiert, ist

$$N_2 = (\{p_0, p_1, p_2\}, \{\mathtt{0}, \mathtt{1}\}, \delta_2, p_0, \{p_2\})$$

und die Übergangsrelation δ_2 ist

	0	1
p_0	$\emptyset$	$\{p_1\}$
p_1	$\emptyset$	$\{p_2\}$
p_2	$\emptyset$	$\emptyset$

Das Zustandsdiagrammm ist



Der Produktautomat N, der die Sprache $L_1 \cup L_2$ akzeptiert, ist

$$N = (Q, \Sigma, \delta, q^s, F)$$

mit

$$\begin{aligned} Q &= Q_1 \cup Q_2 \cup \{q_n\} \\ &= \{q_0, q_1, p_0, p_1, p_2, q_n\} \end{aligned}$$

hierbei ist q_n ein neuer Zustand, der Startzustand des Produktautomaten ist: $q^s = q_n$. Die Menge der Acceptzustände ist

$$F = F_1 \cup F_2 = \{q_1, p_2\}.$$

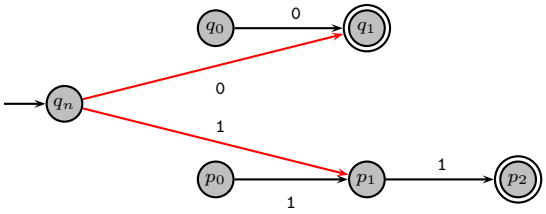
Die Übergangsrelation ist

$$\begin{aligned} \delta = \delta_1 \cup \delta_2 \\ \cup \{(q_n, a, q) \mid (q_1^s, a, q) \in \delta_1\} \\ \cup \{(q_n, a, q) \mid (q_2^s, a, q) \in \delta_2\} \quad &= \delta_1 \cup \delta_2 \cup \{(q_n, \mathtt{0}, q_1\} \cup \{(q_n, \mathtt{1}, p_1)\} \end{aligned}$$

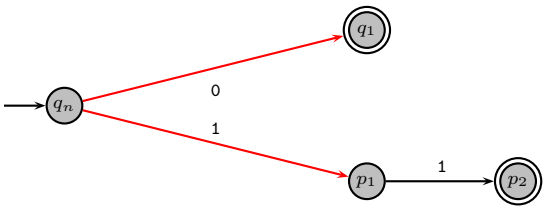
oder in Form einer Tabelle

	0	1
q_n	$\{q_1\}$	$\{p_1\}$
q_0	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\emptyset$
p_0	$\emptyset$	$\{p_1\}$
p_1	$\emptyset$	$\{p_2\}$
p_2	$\emptyset$	$\emptyset$

Das Zustandsdiagramm dieses Produktautomten ist



Dieser Automat kann noch weiter vereinfacht werden, denn die beiden Zustände q_0 und p_0 können vom Startzustand nicht erreicht werden. Daher kann man diese beiden Zustände einfach weglassen.



(c) Die Sprache, die der reguläre Ausdruck

$$\alpha_3 = (\texttt{01})^*.$$

beschreibt, besteht aus allen Bitstrings, die den Teilstring `01` beliebig oft konkatenieren, *i.e.*

$$L(\alpha_3) = \{\lambda, \texttt{01}, \texttt{0101}, \texttt{010101}, \ldots\}.$$

Die Sprache $L = \{\texttt{01}\}$ wird durch den `NFA`

$$N_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$

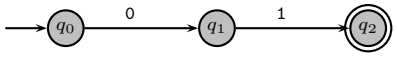
akzeptiert mit

$$Q_1 = \{q_0, q_1, q_2\}, \Sigma = \{\texttt{0}, \texttt{1}\}, q_1^s = q_0, F_1 = \{q_2\}$$

und der Übergangsrelation

	0	1
q_0	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

Das Zustandsdiagramm ist



Wir konstruieren nun den NFA, der die Sprache L^* erkennt. Das allgemeine Procedere lautet (Satz [3.16])

Es gibt einen NFA

$$N = (Q, \Sigma, \delta, q^s, F)$$

mit $L(N) = L(\alpha_1)$,definiert wie folgt: Da $q_1^s \notin F_1$ wählt man einen neuen Startzustand $q_n \notin Q_1$ mit $q^s = q_n$

$$Q = Q_1 \cup \{q_n\}, q^s = q_n, F = F_1 \cup \{q_n\},$$

und

$$\begin{aligned}\delta = \delta_1 \\ &\cup \{(p, a, q_n) \mid (p, a, q) \in \delta_1, q \in F_1\} \\ &\cup \{(q_n, a, q) \mid (q_1^s, a, q) \in \delta_1\} \\ &\cup \{(q_n, a, q_n) \mid (q_1^s, a, q) \in \delta_1, q \in F_1\}\end{aligned}$$

Also: Die menge der Zustände ist:

$$Q = \{q_n, q_0, q_1, q_2\}, q^s = q_n, F = \{q_n, q_2\}$$

Die Übergangsrelation δ ist δ_1 erweitert durch die folgenden Übergänge:

$$\text{Weil } (q_1, 1, q_2) \in \delta_1, q_2 \in F_1 \Longrightarrow (q_1, 1, q_n) \in \delta$$

oder in der üblichen Notation:

$$\text{weil } \delta_1(q_1, 1) = q_2 \in \delta_1, q_2 \in F_1 \Longrightarrow \delta(q_1, 1) = q_n.$$

Weiterhin:

$$\text{Weil } (q_0, 0, q_1) \in \delta_1 \Longrightarrow (q_n, 0, q_1) \in \delta$$

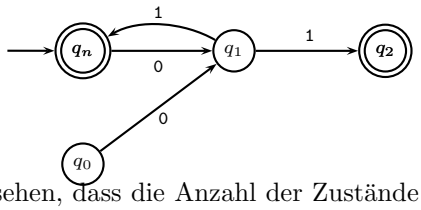
oder in der üblichen Notation:

$$\text{weil } \delta_1(q_0, 0) = q_1 \in \delta_1 \Longrightarrow \delta(q_n, 0) = q_1.$$

Damit erhalten wir die folgende Relation:

	0	1
q_n	$\{q_1\}$	$\emptyset$
q_0	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\{q_2, q_n\}$
q_2	$\emptyset$	$\emptyset$

Der Automat N hat das folgende Zustandsdiagramm:



Es ist leicht zu sehen, dass die Anzahl der Zustände dieses NFA reduziert werden kann, ohne dass sich die Sprache ändert, die der Automat akzeptiert. So ist q_0 irrelevant, da dieser Zustand nicht vom Startzustand aus ereichbar ist. Weiterhin können q_2 und q_n zusammengefasst werden.

A.3.17 Übung [3.37]

Betrachten Sie die folgende rechtslineare Grammatik G:

$$G = (N, T, P, S)$$

mit $N = \{S, A, B\}, T = \mathtt{0}, \mathtt{1}$ und den Produktionen

$$\begin{aligned} S &\longrightarrow \mathtt{0}S \mid \mathtt{1}S \mid \mathtt{0}A \\ A &\longrightarrow \mathtt{0}B \mid \mathtt{1}B \\ B &\longrightarrow \mathtt{0} \mid \mathtt{1}. \end{aligned}$$

Konstruieren Sie einen äquivalenten NFA, indem Sie das Verfahren aus dem Satz [3.9] anwenden.

Lösung:

In einem ersten Schritt ändern wir die Grammatik G ab, wir fügen einen weiteren Zustand E ein, entfernen die Regeln $B \longrightarrow \mathtt{0} \mid \mathtt{1}$ und fügen die Folgenden Regeln hinzu:

$$\begin{aligned} B &\longrightarrow \mathtt{0}E \mid \mathtt{1}E \\ E &\longrightarrow \lambda. \end{aligned}$$

Damit wir die geänderte Grammatik:

$$G' = (N', T, P', S)$$

mit $N = \{S, A, B, E\}, T = \mathtt{0}, \mathtt{1}$ und den Produktionen

$$\begin{aligned} S &\longrightarrow \mathtt{0}S \mid \mathtt{1}S \mid \mathtt{0}A \\ A &\longrightarrow \mathtt{0}B \mid \mathtt{1}B \\ B &\longrightarrow \mathtt{0}E \mid \mathtt{1}E \\ E &\longrightarrow \lambda. \end{aligned}$$

Aus der Grammatik G' lässt sich ein NFA wie folgt erstellen:

$$N = (Q, \Sigma, \delta, q^s, F)$$

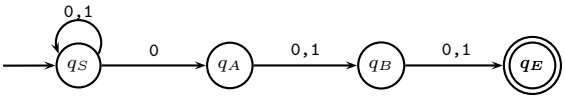
mit

$$Q = \{q_S, q_A, q_B, q_E\}, \quad \Sigma = \{\mathtt{0}, \mathtt{1}\}$$

dem Startzustand $q^s = q_S$, der Menge der Acceptzustände $F = \{q_E\}$ und der Übergangsrelation δ:

	0	1
q_S	$\{q_S, q_A\}$	$\{q_S\}$
q_A	$\{q_B\}$	$\{q_B\}$
q_B	$\{q_E\}$	$\{q_E\}$
q_E	$\emptyset$	$\emptyset$

Das Zustandsdiagramm dieses Automaten sieht folgendermaßen aus:



A.3.18 Übung [3.38]

Betrachten Sie die reguläre Sprache

$$L = \{w \in \{\texttt{0},\texttt{1}\}^* \mid w \text{ enthält eine gerade Anzahl von } \texttt{0}\text{en und } \texttt{1}\text{en}\}.$$

1. Erstellen Sie einen DFA, der diese Sprache akzeptiert.
2. Konstruieren Sie aus diesem Automaten eine rechtslineare Grammatik, die die Sprache L erzeugt.

Lösung:

Der folgende `DFA` akzeptiert sämtliche Bitstrings, die sowohl eine gerade Anzal von `0`en als auch eine gerade Anzahl von `1`en enthalten.

$$M = (Q, \Sigma, \delta, q^s, F)$$

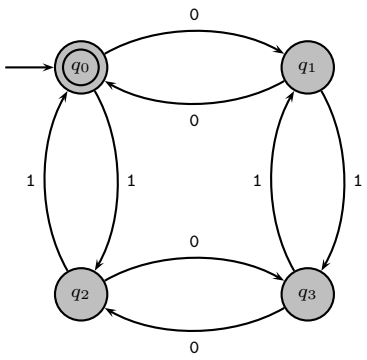
mit

$$Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{\texttt{0},\texttt{1}\}, q^s = q_0, F = \{q_0\},$$

und der Übergangsfunktion δ:

	`0`	`1`
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_1

Das Zustandsdiagramm ist:



Wir konstruieren eine rechtslineare Grammatik, die die gleiche Sprache L erzeugt. Es ist

$$G = (N, T, P, S).$$

Jedem Zustand q des Automaten M ordnen wir eine Variable zu,

$$\begin{array}{lcl} q_0 & \longrightarrow & X_0, \\ q_1 & \longrightarrow & X_1, \\ q_2 & \longrightarrow & X_2, \\ q_3 & \longrightarrow & X_3. \end{array}$$

Damit

$$N = \{X_0, X_1, X_2, X_3\},$$

das Terminalzeichen Alphabet ist $T = \Sigma = \{\mathtt{0}, \mathtt{1}\}$, die Startvariable ist X_0. Die Produktionen sind

$$\begin{array}{lcl} X_0 & \longrightarrow & \mathtt{0}X_1 \mid \mathtt{1}X_2 \mid \lambda \\ X_1 & \longrightarrow & \mathtt{0}X_0 \mid \mathtt{1}X_3 \\ X_2 & \longrightarrow & \mathtt{0}X_3 \mid \mathtt{1}X_0 \\ X_3 & \longrightarrow & \mathtt{0}X_2 \mid \mathtt{1}X_1. \end{array}$$

A.3.19 Übung [3.39]

Betrachten Sie die reguläre Sprache

$$L = \{w \in \{0,1\}^* \mid w \text{ enthält nicht den Teilstring } 110\}.$$

1. Konstruieren Sie einen DFA, der L akzeptiert.
2. Konstruieren Sie aus dem DFA den regulären Ausdruck, der die Sprache L generiert.

Lösung:

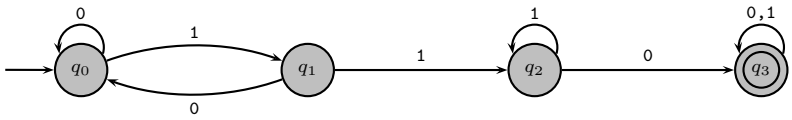
Wir konstruieren zunächst einen DFA, der die Sprache

$$L = \{w \in \{0,1\}^* \mid w \text{ enthält nicht den Teilstring } 110\}$$

akzeptiert. Dazu konstrieren wir den DFA, der die Komplementsprache akzeptiert und vertauschen die Accept– und Nichtacceeptzustände.

Der folgende Automat akzeptiert die Sprache

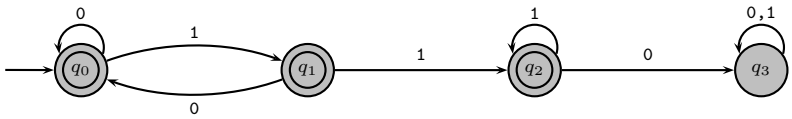
$$\overline{L} = \{w \in \{0,1\}^* \mid w \text{ enthält den Teilstring } 110\}.$$



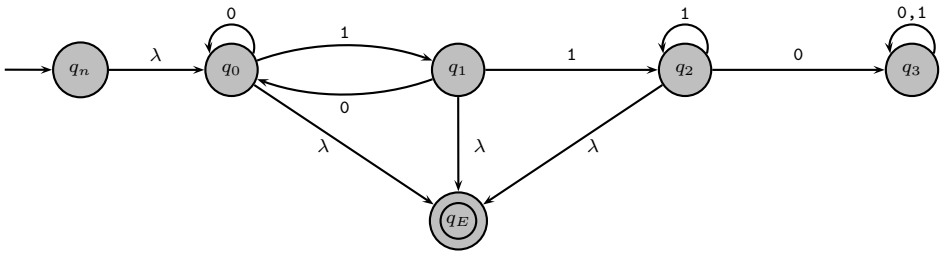
Vertauscht man nun die Acceptzustände und die Nicht–Acceptzustände, ergibt sich der DFA, der die Komplementsprache akzeptiert, *i.e.* die Sprache

$$L = \{w \in \{0,1\}^* \mid w \text{ enthält nicht den Teilstring } 110\}.$$

Damit erhalten wir den folgenden DFA:



Die Kontruktion des regulären Ausdrucks, der mit diesem `DFA` korrespondiert, beginn damit, dass man einen neuen Startzustand q_n hinzufügt, der mit einem λ– Übergang in den alten Startzustand übergeht. Weiterhin wird ein neuer Endzustand q_E eingefügt. Alle bisherigen Acceptzustände verlieren diese Eigenschaft, es werden λ–Übergänge von den alten Acceptzustände in den q_E Zustand eingefügt. Dies führt auf den folgenden Automaten.



In diesem Automaten werden nun alle internen Zustände eliminiert, dabei werden die Übergänge angepasst.

Elimination des Zustands q_1

Wir betrachten alle möglichem Kanten, die den Zustand q_1 als Zwischenzustand enthalten.

1. Wir haben den Teilgraphen

$$q_0 \xrightarrow{\texttt{1}} q_1 \xrightarrow{\texttt{0}} q_0.$$

Die Elimination von q_1 führt zu einem Loop am Zustand q_0 mit der Beschriftung `10`.

2. Wir haben den Teilgraphen

$$q_0 \xrightarrow{\texttt{1}} q_1 \xrightarrow{\texttt{1}} q_2.$$

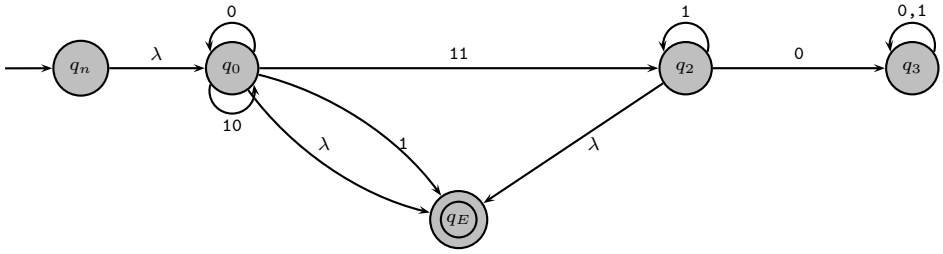
Wird der Zustand q_1 eliminiert, führt dies auf einen Übergang von q_0 nach q_2 mit der Beschriftung `11`.

3. Schließlich haben wir den Teilgraphen

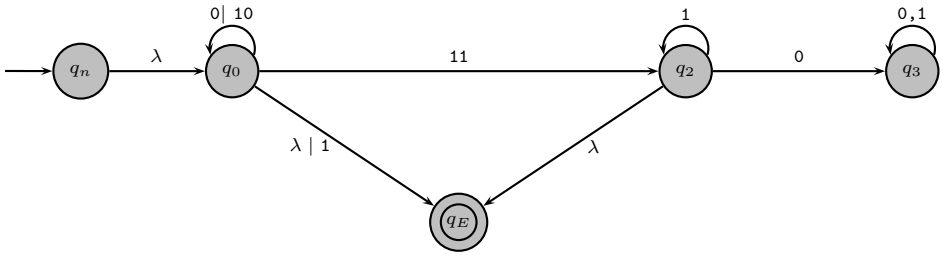
$$q_0 \xrightarrow{\texttt{1}} q_1 \xrightarrow{\lambda} q_E.$$

Bei der Elimination von q_1 haben wir einen Übergang von q_0 nach q_E hinzuzufügen mit der Beschriftung `1`.

Weitere Kanten gibt es nicht. Eliminiert man den Zustand q_1, erhalten wir damit den folgenden Graphen:



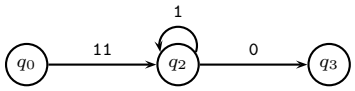
Wir ersetzen die goppelten Übergänge durch einfache und beschriften als Alternative.



Elimination des Zustands q_2

Wir betrachten alle möglichem Kanten, die den Zustand q_2 als Zwischenzustand enthalten.

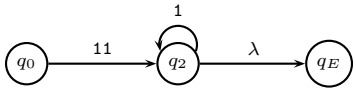
1. Wir haben den Teilgraphen



Durch die Elimination von q_2 haben wir einen Übergang von q_0 nach q_3 mit der Beschriftung

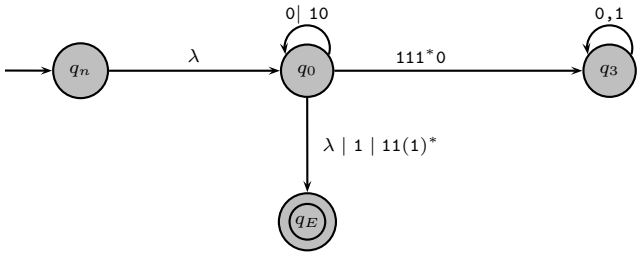
$$111^*0$$

2. Wir haben den Teilgraphen



Wird der Zustand q_2 eliminiert, führt dies auf einen Übergang von q_0 nach q_E mit der Beschriftung $\mathtt{111}^*$.

Damit erhalten wir:



Elimination des Zustands q_0

Schließlich müssen wir noch den Zustand q_0 eliminieren. Es gibt zwei Kantenzüge

(a) $q_n \longrightarrow q_0 \longrightarrow q_3$
Wird hier q_0 eliminiert, erhält man einen Übergang von q_n in q_3 mit Beschriftung

$$(\mathtt{0} \mid \mathtt{10})^*\mathtt{11(1)}^*\mathtt{0}.$$

Für die Ableitung des regulären Ausdrucks ist dieser Übergang nicht relevant, da der Endzustand q_3 kein Acceptzustand ist, es können auch keine weiteren Zustände mehr eliminiert werden.

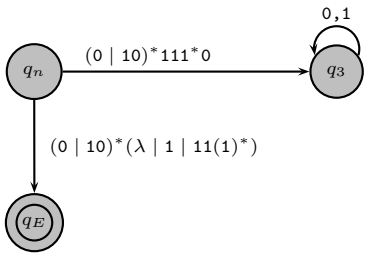
(b) Für die Herleitung des regulären Ausdrucks ist der Übergang

$$q_n \longrightarrow q_0 \longrightarrow q_E$$

relevant. Die Elimination von q_0 führt auf einen Übergang $q_n \longrightarrow q_E$, die Beschriftung ist

$$(\mathtt{0} \mid \mathtt{10})^* \mid \lambda \mid \mathtt{1} \mid \mathtt{11(1)}^*$$

Damit ergibt sich der finale Graph zu:



Die Beschriftung des Übergangs $q_n \longrightarrow q_E$ im finalen Graph ist der gesuchte reguläre Ausdruck, *i.e.*

$$R = (\mathtt{0} \mid \mathtt{10})^*(\lambda \mid \mathtt{1} \mid \mathtt{11}(\mathtt{1})^*).$$

A.3.20 Übung [3.40]

Betrachten Sie die folgende Sprache

$$L = \{w \in \{a,b\}^* \mid w \text{ beginnt oder endet mit } aa \text{ oder } bb\}.$$

Zeigen Sie, dass L regulär ist. Konstruieren Sie zu diesem Zweck einen `DFA` mit der in Abschnitt [3.9.1] vorgestellten Methode. Lassen Sie nicht erreichbare Zustände unberücksichtigt.

Lösung:

Wir betrachten die vier Sprachen

$$\begin{aligned}
L_1 &= \{w \in \{a,b\}^* \mid w = aav, v \in \{a,b\}^*\},\\
L_2 &= \{w \in \{a,b\}^* \mid w = vaa, v \in \{a,b\}^*\},\\
L_3 &= \{w \in \{a,b\}^* \mid w = bbv, v \in \{a,b\}^*\},\\
L_4 &= \{w \in \{a,b\}^* \mid w = vbb, v \in \{a,b\}^*\}.
\end{aligned}$$

Dann können wir die Sprache

$$L = \{w \in \{a,b\}^* \mid w \text{ beginnt oder endet mit } aa \text{ oder } bb\}$$

als Vereinigung

$$L = L_1 \cup L_2 \cup L_3 \cup L_4$$

schreiben. Um den Automaten zu konstruieren, der die Sprache L akzeptiert, erstellen wir zunächst die Automaten, die $L_1 \cup L_2$ und $L_3 \cup L_4$ erkennen. Aus diesen beiden Automaten konstruieren wir schließlich den `DFA`, der L akzeptiert.

Der `DFA` M_1, der L_1 akzeptiert ist

$$M_1 = (Q_1, \Sigma, \delta_1, q_1^s, F_1)$$

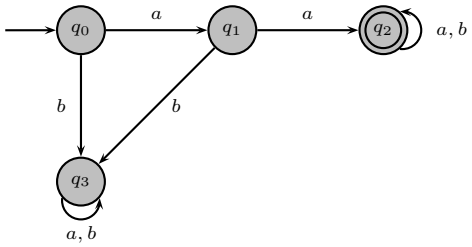
mit

$$Q_1 = \{q_0, q_1, q_2, q_3\}$$

Startzustand ist q_0, Acceptzustand ist q_2 und die Übergangsfunktion ist

	a	b
q_0	q_1	q_3
q_1	q_2	q_3
q_2	q_2	q_2
q_3	q_3	q_3

Das Zustandsdiagramm ist:



Der DFA M_2, der die Sprache L_2 akzeptiert ist

$$M_2 = (Q_2, \Sigma, \delta_2, q_2^s, F_2)$$

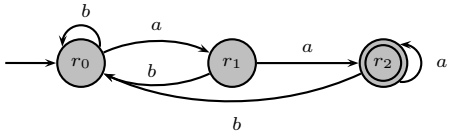
mit

$$Q_2 = \{r_0, r_1, r_2\}$$

Startzustand ist r_0, Acceptzustand ist r_2 und die Übergangsfunktion ist

	a	b
r_0	r_1	r_0
r_1	r_2	q_0
r_2	r_2	q_0

Das Zustandsdiagramm ist:



Es ist nun sehr einfach, die Automaten für die Sprachen L_3 und L_4 zu erstellen.
Der DFA M_3, der L_3 akzeptiert ist

$$M_3 = (Q_3, \Sigma, \delta_3, q_3^s, F_3)$$

mit

$$Q_3 = \{p_0, p_1, p_2, p_3\}$$

Startzustand ist p_0, Acceptzustand ist p_2 und die Übergangsfunktion ist

	a	b
p_0	p_3	p_1
p_1	p_3	p_2
p_2	p_2	p_2
p_3	p_3	p_3

Das Zustandsdiagramm ist mit dem des DFA M_1 identisch, es muss nur a udn b vertauscht werden.

Der DFA M_4, der die Sprache L_4 akzeptiert ist

$$M_4 = (Q_4, \Sigma, \delta_4, q_4^s, F_4)$$

mit

$$Q_4 = \{s_0, s_1, s_2\}$$

Startzustand ist s_0, Acceptzustand ist s_2 und die Übergangsfunktion ist

	a	b
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_2

Für das Zustandsdiagramm gilt das Gleiche wie oben.

Wir konstruieren nun aus den DFA M_1 und M_2 den DFA, der die Sprache $L_1 \cup L_2$ akzeptiert, *i.e.* die Menge aller Strings, die entweder mit aa beginnen oder mit aa enden. Dieser Automat ist:

$$M_5 = (Q_5, \Sigma, \delta_5, q_5^s, F_5)$$

mit

$$\begin{aligned} Q_5 &= Q_1 \times Q_2 \\ &= \{q_0, q_1, q_2, q_3\} \times \{r_0, r_1, r_2\} \\ &= \{(q_0, r_0), (q_0, r_1), (q_0, r_2), (q_1, r_0), (q_1, r_1), (q_1, r_2), \\ &\quad \{(q_2, r_0), (q_2, r_1), (q_2, r_2), (q_3, r_0), (q_3, r_1), (q_3, r_2)\} \end{aligned}$$

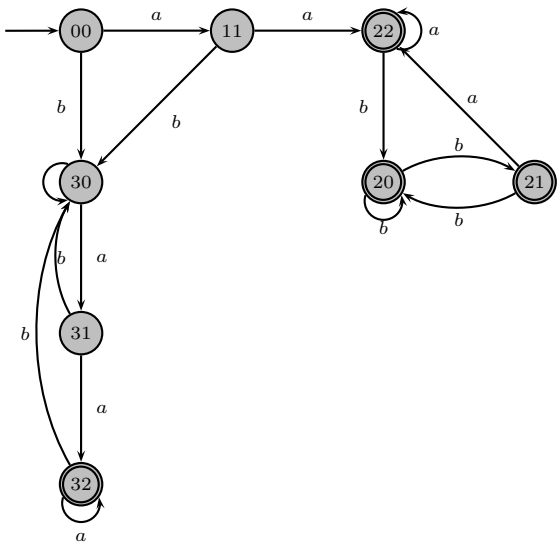
Startzustand: $q_5^s = (q_0, r_0)$
Menge der Acceptzustände

$$\begin{aligned} F_5 &= F_1 \times Q_2 \cup Q_1 \times F_2 \\ &= \{q_2\} \times Q_2 \cup Q_1 \times \{r_2\} \\ &= \{(q_2, r_0), (q_2, r_1), (q_2, r_2), (q_0, r_2), (q_1, r_2), (q_3, r_2)\}. \end{aligned}$$

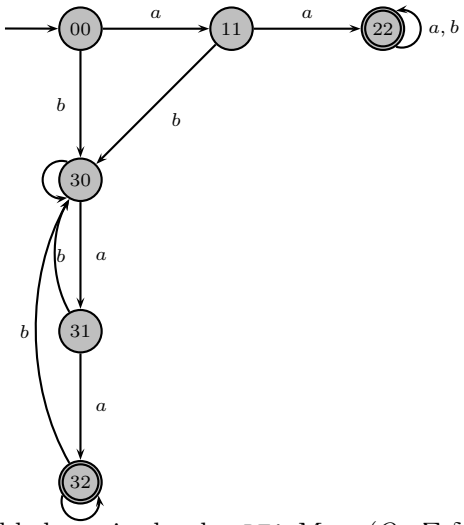
Die Übergangsfunktion δ_5 ist:

	a	b
(q_0, r_0)	(q_1, r_1)	(q_3, r_0)
(q_0, r_1)	(q_1, r_2)	(q_3, r_0)
(q_0, r_2)	(q_1, r_2)	(q_3, r_0)
(q_1, r_0)	(q_2, r_1)	(q_3, r_0)
(q_1, r_1)	(q_2, r_2)	(q_3, r_0)
(q_1, r_1)	(q_2, r_2)	(q_3, r_0)
(q_2, r_0)	(q_2, r_1)	(q_2, r_0)
(q_2, r_1)	(q_2, r_2)	(q_2, r_0)
(q_2, r_2)	(q_2, r_2)	(q_2, r_0)
(q_3, r_0)	(q_3, r_1)	(q_3, r_0)
(q_3, r_1)	(q_3, r_2)	(q_3, r_0)
(q_3, r_2)	(q_3, r_2)	(q_3, r_0)

Das Zustandsdiagramm erhalten wir folgendermaßen. Man beginnt im Startzustand (q_0, r_0) und betrachtet mit der Übergangsfunktion alle Zustände, die sich vom Startzustand erreichen lassen. Daraus ergibt sich das folgende Zustandsdiagramm. Der Einfachheit halber bezeichnen wir die Zustände (q_i, r_j) mit ij.



Dieses Zustandsdiagramm kann noch vereinfacht werden, denn ist der Automat einmal im Zustand 22, dann bleibt er in $22, 20$ oder 21 Alle drei Zustände sind Acceptzustände, die zu einem Zustand zusammengzogen werden können. damit erhalten wir den Minimalautomaten, der die Sprachem $L_1 \cup L_2$ akzeptiert:



Formal haben wir also den DFA $M_5 = (Q_5, \Sigma.\delta_5, q_5^s, F_5)$, der die Sprache

$$L_1 \cup L_2 = \{w \in \{a,b\}^* \mid w = aav \text{ oder } w = vaa\}$$

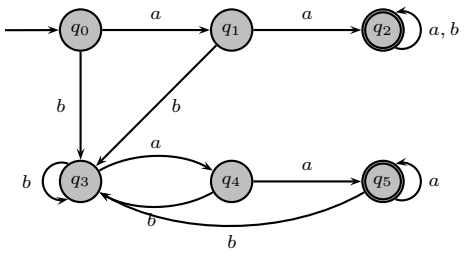
akzeptiert, gegeben durch sechs Zustände

$$Q_5 = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

Startzustand ist q_0, Acceptzustände sind $F_5 = \{q_2, q_5\}$, Übergangsfunktion δ_5 ist

	a	b
q_0	q_1	q_3
q_1	q_2	q_3
q_2	q_2	q_2
q_3	q_4	q_3
q_4	q_5	q_3
q_5	q_5	q_3

Das Zustandsdiagramm ist:



Analog ergibt sich der Automat, der die Sprache

$$L_3 \cup L_4 = \{w \in \{a,b\}^* \mid w = bbv \text{ oder } w = vbb\}$$

akzeptiert durch vertauschen der a und b Übergänge.

$$M_6 = (Q_6, \Sigma, \delta_6, q_6^s, F_6)$$

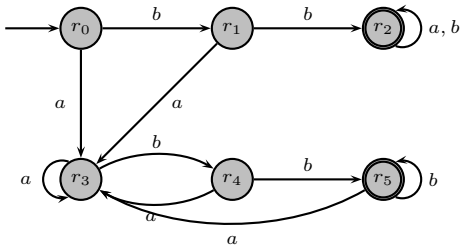
mit den sechs Zuständen

$$Q_6 = \{r_0, r_1, r_2, r_3, r_4, r_5\},$$

dem Startzustand r_0; die Acceptzustände sind $F_6 = \{r_2, r_5\}$, Übergangsfunktion δ_6 ist

	a	b
r_0	r_1	r_1
r_1	r_3	r_2
r_2	r_2	r_2
r_3	r_3	r_4
r_4	r_3	r_5
r_5	r_3	r_5

Das Zustandsdiagramm ist:



Wir konstruieren nun aus den Automaten M_5 und M_6 den Produktautomaten M, der die Sprache

$$L = \{w \in \{a,b\}^* \mid w \text{ beginnt oder endet mit } aa \text{ oder } bb\}$$

akzeptiert. Wir haben

$$M = (Q, \Sigma, \delta, q^s, F)$$

mit den 36 Zuständen

$$\begin{aligned} Q &= Q_5 \times Q_6 \\ &= \{(q_0,r_0),(q_0,r_1),(q_0,r_2),(q_0,r_3),(q_0,r_4),(q_0,r_5), \\ &\quad (q_1,r_0),(q_1,r_1),(q_1,r_2),(q_1,r_3),(q_1,r_4),(q_1,r_5), \\ &\quad (q_2,r_0),(q_2,r_1),(q_2,r_2),(q_2,r_3),(q_2,r_4),(q_2,r_5), \\ &\quad (q_3,r_0),(q_3,r_1),(q_3,r_2),(q_3,r_3),(q_3,r_4),(q_3,r_5), \\ &\quad (q_4,r_0),(q_4,r_1),(q_4,r_2),(q_4,r_3),(q_4,r_4),(q_4,r_5), \\ &\quad (q_5,r_0),(q_5,r_1),(q_5,r_2),(q_5,r_3),(q_5,r_4),(q_5,r_5)\}\,. \end{aligned}$$

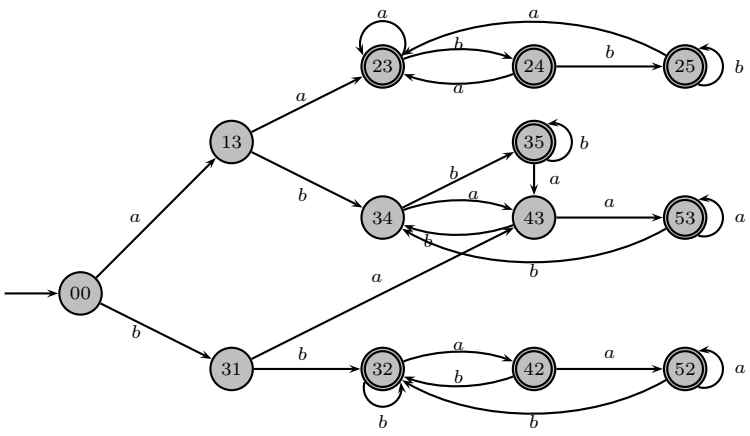
Der Startzustand ist $q^s = (q_0, r_0)$, die Menge der Acceptzustände ist gegeben durch

$$\begin{aligned} F &= (F_5 \times Q_6) \cup (Q_5 \times F_6) \\ &= (\{q_2, q_5\} \times Q_6) \cup (Q_5 \times \{r_2, r_5\}) \\ &= \{(q_2, r_0), (q_2, r_1), (q_2, r_2), (q_2, r_3), (q_2, r_4), (q_2, r_5) \\ &\quad (q_5, r_0), (q_5, r_1), (q_5, r_2), (q_5, r_3), (q_5, r_4), (q_5, r_5) \\ &\quad (q_0, r_2), (q_1, r_2), (q_3, r_2), (q_4, r_2), (q_5, r_2) \\ &\quad (q_0, r_5), (q_1, r_5), (q_2, r_5), (q_3, r_4), (q_4, r_5)\} . \end{aligned}$$

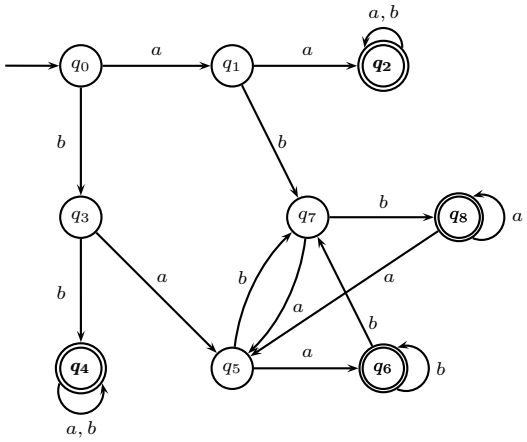
Die Übergangsfunktion δ ist: (Wir verwenden wieder die Kurzschreibweise ij für den Zustand (q_i, r_j) für $i, j = 0, \ldots, 5$)

	a	b			a	b
00	13	31		30	43	31
01	13	32		30	43	32
02	12	32		30	42	32
03	13	34		30	43	34
04	13	35		30	43	35
05	13	35		30	43	35
10	23	31		40	53	31
11	23	32		40	53	32
12	22	32		40	52	32
13	23	34		40	53	34
14	23	35		40	53	35
15	23	35		40	53	35
20	23	21		50	53	31
21	23	22		50	53	32
22	22	22		50	52	32
23	23	24		50	53	34
24	23	25		50	53	35
25	23	25		50	53	35

Um das Zustandsdiagramm zu erhalten, beginnen wir wieder im Startzustand (q_0, r_0) und betrachten die Zustände, die von hieraus angenommen werden. Daraus ergibt sich der folgende Graph:



Diesen Automaten kann man noch vereinfachen. Die Zustände 23, 24 und 25 können zusammengezogen werden, ebenfalls die drei Zustände 32, 42, 52.



A.3.21 Übung [3.41]

Minimieren Sie den folgenden DFA M:

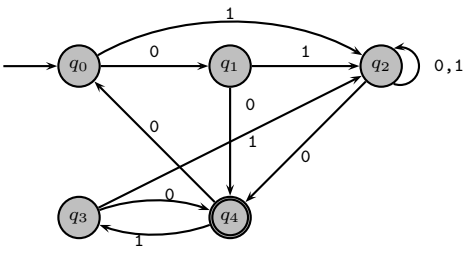
$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, q_0, \{q_4\})$$

mit der Übergangsfunktion

	0	1
q_0	q_1	q_2
q_1	q_4	q_2
q_2	q_4	q_2
q_3	q_4	q_2
q_4	q_0	q_3

Lösung:

Das Zustandsdiagramm dieses DFA ist



Um diesen Automaten zu minimieren, erstellen wir zunächst die initiale Tabelle.

q_0				
—	q_1			
—	—	q_2		
—	—	—	q_3	
—	—	—	—	q_4

Da q_4 der einzige Acceptzustand ist, sind initial die Zustandspaare

$$\{q_0, q_4\}, \{q_1, q_4\}, \{q_2, q_4\}, \{q_3, q_4\}$$

in der Tabelle zu markieren. Damit ist

q_0				
—	q_1			
—	—	q_2		
—	—	—	q_3	
X_0	X_0	X_0	X_0	q_4

1. Iteration

Betrachte die nicht markierten Paare

$$\begin{array}{lcll}
\{q_0,q_1\} & \xrightarrow{0} & \{q_1,q_4\} & \text{markiert}\\
\{q_0,q_1\} & \xrightarrow{1} & -- & \\
\{q_0,q_2\} & \xrightarrow{0} & \{q_1,q_4\} & \text{markiert}\\
\{q_0,q_2\} & \xrightarrow{1} & -- & \\
\{q_1,q_2\} & \xrightarrow{0,1} & -- & \\
\{q_0,q_3\} & \xrightarrow{0} & \{q_1,q_4\} & \text{markiert}\\
\{q_0,q_3\} & \xrightarrow{1} & -- &
\end{array}$$

Alle anderen Paare liefern keinen markierten Übergang. Damit ist nach der 1. Runde

$$\begin{array}{lllll}
q_0 & & & & \\
X_1 & q_1 & & & \\
X_1 & — & q_2 & & \\
X_1 & — & — & q_3 & \\
X_0 & X_0 & X_0 & X_0 & q_4
\end{array}$$

Es sind keine weiteren Markierungen möglich. Damit sind die drei Zustände q_1, q_2 und q_3 äquivalent und können zu einem Zustand zusammengezogen werden.

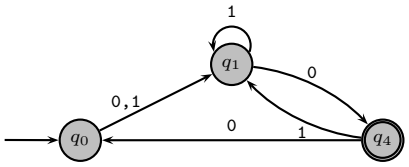
Der Minimalautomat ist daher

$$M_{min} = (\{q_0,q_1,q_4\},\{0,1\},\delta,q_0,\{q_4\})$$

mit der Übergangsfunktion

	0	1
q_0	q_1	q_2
q_1	q_4	q_2
q_4	q_0	q_3

Das Zustandsdiagramm ist:



A.4 Lösungen zu den Übungen aus Kapitel [4.8]

A.4.1 Übung [4.42]

Geben Sie eine kontextfreie Grammatik G an, für die gilt: $L = L(G)$ mit

$$L = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w = \mathtt{0}^i\mathtt{1}^j, i < j, i, j \in \mathbb{N}_0\}.$$

Definieren Sie die Grammatik vollständig.

Lösung:

Die Grammatik ist das 4–Tupel

$$G = (N, T, P, S)$$

mit der Menge der Variablen

$$N = \{S\},$$

dem Terminalalphabet

$$T = \{\mathtt{0},\mathtt{1}\},$$

den Produktionen

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}S\mathtt{1} \\ S & \longrightarrow & S\mathtt{1} \\ S & \longrightarrow & \mathtt{1}. \end{array}$$

und der Startvariablen S.

Die erste Produktion generiert stets die gleiche Anzahl von `0`en und `1`en. Damit die Ersetzungen terminieren und ein String aus Terminalzeichen entsteht, muss entweder die zweite und dritte Regel oder nur die dritte Regel angewendet werden. Dadurch entsteht ein Terminalzeichenwort, das mindestens eine `1` mehr hat als `0`en.

A.4.2 Übung [4.43]

Sei M der Pushdown Automat, definiert durch

$Q = \{q_1, q_2, q_3, q_4\}$,

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

$\Gamma = \{A, \bot\}$,

$q_s = q_1$

$F = \{q_3, q_4\}$.

und

$$\begin{aligned}
\delta(q_1, a, \bot) &= \{(q_1, A\bot)\} \\
\delta(q_1, a, A) &= \{(q_1, AA)\} \\
\delta(q_1, b, A) &= \{(q_2, \lambda)\} \\
\delta(q_1, \lambda, A) &= \{(q_4, A)\} \\
\delta(q_1, \lambda, \bot) &= \{(q_4, \bot)\} \\
\delta(q_2, b, A) &= \{(q_2, \lambda)\} \\
\delta(q_2, \lambda, A) &= \{(q_3, A)\}. \\
\delta(q_2, \lambda, \bot) &= \{(q_3, \bot)\}. \\
\delta(q_4, a, \bot) &= \{(q_4, \bot)\}.
\end{aligned}$$

(a) Man beschreibe die Sprache L, die von M akzeptiert wird.

(b) Man gebe ein Zustandsdiagramm des Automaten an.

(c) Man erstelle die Ableitungen der Strings `aab`, `abb` und `aba` in M.

(d) Man zeige, dass `aabb`, `aaab` $\in L(M)$.

Lösung:

(a) Die Sprache, die von dem Pushdown Automaten M akzeptiert wird, ist

$$L = \{a^i b^j, 0 \leq j \leq i\}.$$

- Der Zustand q_1 zählt die führenden `a`s.
- Sobald das erste `b` gelesen wird — falls vorhanden, geht der PDA in den Zustand q_2 über und nimmt ein Marker vom Stack.
- Im Zustand q_2 werden die restlichen `b`s gezählt, *i.e.* für jedes `b` des Eingabestrings wird ein Marker vom Stack genommen.

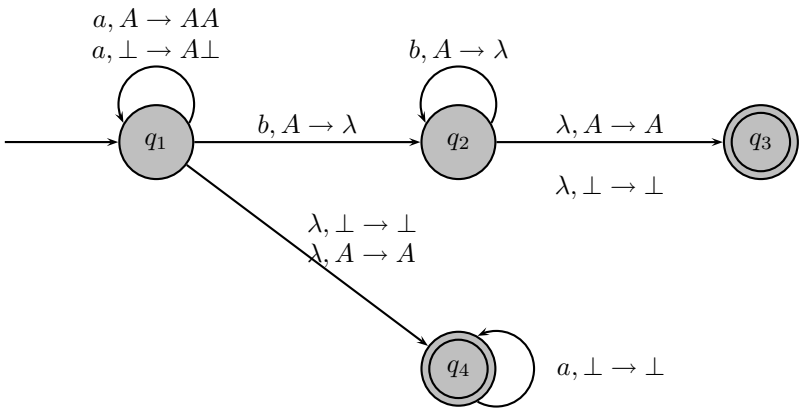
- Ist der String abgearbeitet, *i.e.* der zu verarbeitende String ist λ, *und* liegt auf dem Stack noch ein Marker, dann hat der ursprüngliche String mit Sicherheit weniger `b`s als `a`s. Dann geht der PDA in den Acceptzustand q_3 über und läßt den Stack unverändert.
- Ist der String abgearbeitet, *i.e.* der zu verarbeitende String ist λ, *und* auf dem Stack liegt der Begrenzer $\perp$, dann hat der ursprüngliche String genauso viele `a`s wie `b`s, dieser String ist also ebenfalls zu akzeptieren, daher geht die Maschine ebenfalls in den Zustand q_3 über.
- Der Automat akzeptiert auch Strings der Form $w = a^n, n = 0, 1, 2, \ldots$, also insbesondere auch den leeren String λ. Dazu dient der nichtdeterministische Übergang von q_1 nach q_4 durch

$$\delta(q_1, \lambda, A) = \{(q_4, A)\},$$
$$\delta(q_1, \lambda, \perp) = \{(q_4, \perp)\}.$$

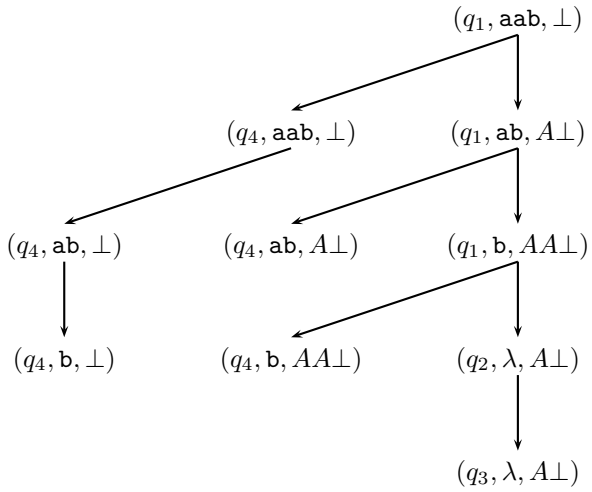
Wenn der Automat ein Inputsymbol `a` liest, wird jedesmal das Symbol `A` auf den Stack gelegt. Strings der Form $\texttt{a}^i$ werden im Zustand q_4 akzeptiert. Der Übergang im Zustand q_4 poppt die Symbole `A` vom Stack nachdem der Input verarbeitet wurde.

Eine Verarbeitung eines Strings der Form $\texttt{a}^i\texttt{b}^j$ geht dann in den Zustand q_2 über, sobald das erste `b` gelesen wird. Um den kompletten Inputstring zu verarbeiten, muß der Stack wenigstens j Symbole `A` enthalten.

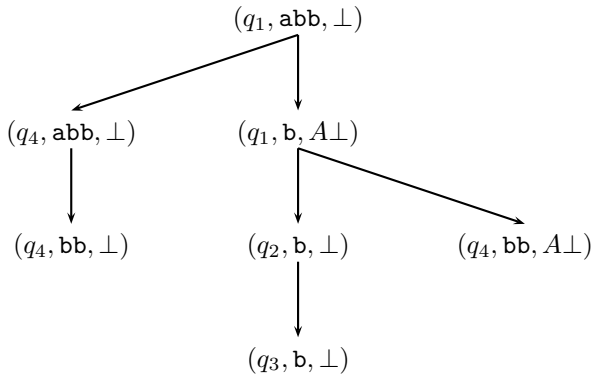
(b) Das Zustandsdigramm dieses Pushdown Automaten ist



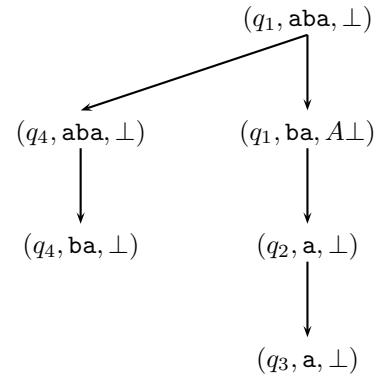
(c) Die Ableitung des Strings `abb` wird folgendermaßen ausgeführt:



Der String `aabb` wird akzeptiert, da der Automat im Zustand q_3 terminiert, der Inputstring ist komplett abgearbeitet, wobei der Stackinhalt keine Rolle spielt.
Die Ableitung des Strings `abb` ist:

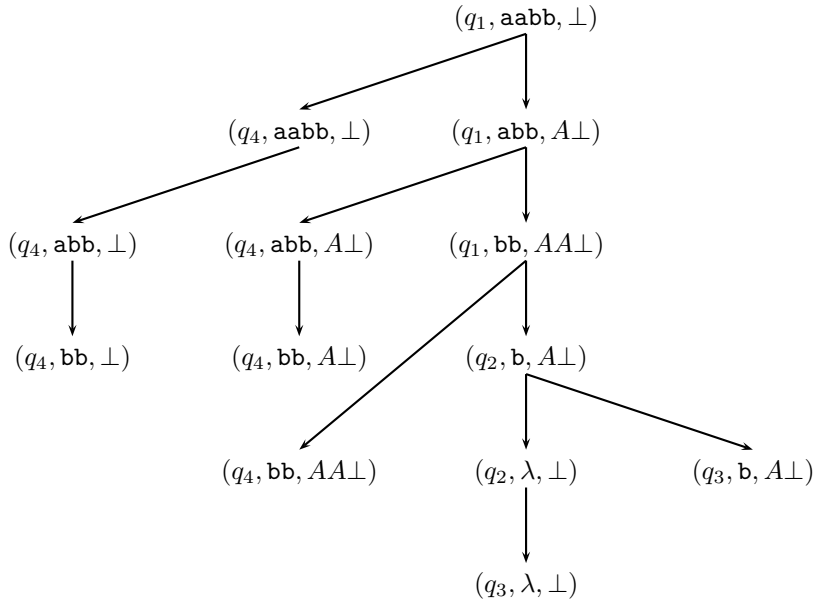


Der String `abb` wird nicht akzeptiert, da alle Verarbeitungszweige mit nichtleeren Inputstrings terminieren.
Der String `aba` wird folgendermaßen abgleitet:

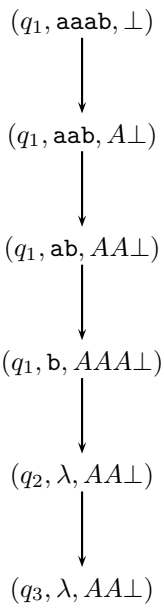


Der String `abb` wird nicht akzeptiert, da alle Verarbeitungszweige mit nichtleeren Inputstrings terminieren.

(d) Die Ableitung des Strings `aabb` durch den PDA ist:



Der String wird akzeptiert, da der PDA im Zustand q_2 terminiert und der Input komplett verarbeitet ist. Die Ableitung des Strings `aaab` ist:



Es sind nicht alle λ–Übergänge eingezeichnet. Der String `AAAb` wird akzeptiert, da der `PDA` sich nach Verarbeitung des kompletten Strings im Acceptzustand q_3 befindet, der Stackinhalt spielt keine Rolle.

A.4.3 Übung [4.44]

Man erstelle Pushdown Automaten für die Sprachen

(a) Die Sprache über dem Alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, deren Worte die gleiche Anzahl von as und bs hat.

(b) Die Sprache über dem Alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, deren Worte die doppelte Anzahl von as wie bs hat.

(c) $\{\omega \mid \omega \in \{a, b\}^*, \omega$ hat doppelt soviele as wie bs$\}$.

(d) $\{a^m b^n, 0 \leq m \leq n \leq 2m\}$.

Geben Sie informelle Beschreibungen an.

Lösung:

(a) Formal kann der PDA, der die Sprache

$$L = \{w \in \{a, b\}^* \mid n_a(w) = n_b(w)\}$$

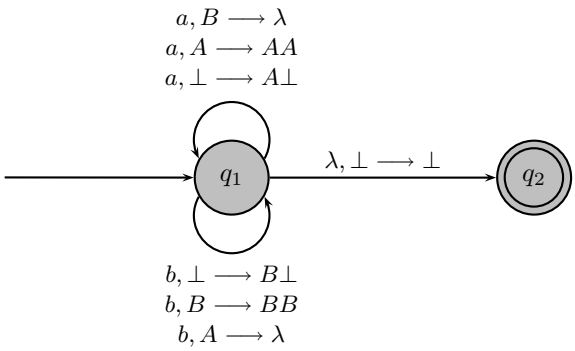
erkennt ($n_a(w)$ bedeutet die Anzahl der a's im String w), beschrieben werden durch das folgende 7–Tupel:

$$P_1 = (Q, \Sigma, \Gamma, \delta, q_s, \bot, F)$$

mit

(a) Der Menge der Zustände $Q = \{q_1, q_2\}$,
(b) dem Inputalphabet $\Sigma = \{a, b\}$,
(c) dem Stackalphabet $\Gamma = \{A, B, \bot\}$,
(d) dem Startzustand $q_s = q_1$,
(e) dem Stackbegrenzer $\bot$
(f) der Menge der Acceptzustände $F = \{q_2\}$
(g) und der Übergangsfunktion

$$\begin{aligned}
\delta(q_1, \mathtt{a}, \bot) &= \{(q_1, \mathtt{A}\bot)\},\\
\delta(q_1, \mathtt{a}, \mathtt{A}) &= \{(q_1, \mathtt{AA})\},\\
\delta(q_1, \mathtt{a}, \mathtt{B}) &= \{(q_1, \lambda)\},\\
\delta(q_1, \mathtt{b}, \bot) &= \{(q_1, \mathtt{B}\bot)\},\\
\delta(q_1, \mathtt{b}, \mathtt{B}) &= \{(q_1, \mathtt{BB})\},\\
\delta(q_1, \mathtt{b}, \mathtt{A}) &= \{(q_1, \lambda)\},\\
\delta(q_1, \lambda, \bot) &= \{(q_2, \bot)\}.
\end{aligned}$$



Das Zustandsdiagramm dieses PDA ist in der folgenden Abbildung skizziert.

Der String `baba` wird folgendermaßen abgearbeitet:

$$\begin{aligned}(q_1, \texttt{baba}, \perp) &\longmapsto (q_1, \texttt{aba}, \texttt{B}\perp)\\ &\longmapsto (q_1, \texttt{ba}, \perp)\\ &\longmapsto (q_1, \texttt{a}, \texttt{B}\perp)\\ &\longmapsto (q_1, \lambda, \perp)\\ &\longmapsto (q_2, \lambda, \perp).\end{aligned}$$

Der Stack zählt die Differenz der gelesenen `a`s und `b`s.

(b) Der `PDA`, der die Sprache

$$L = \big\{\omega \in \{a, b\}^* \mid n_a(\omega) = 2 \times n_b(\omega)\big\}$$

erkennt ist

$$P = \big(Q, \Sigma, \Gamma, \delta, q_s, \perp, F\big)$$

mit

(a) der Menge der Zustände $Q = \{q_1, q_2, q_3\}$,

(b) dem Inputalphabet $\Sigma = \{a, b\}$,

(c) dem Stackalphabet $\Gamma = \{A, B, \perp\}$,

(d) dem Startzustand $q_2 = q_1$,

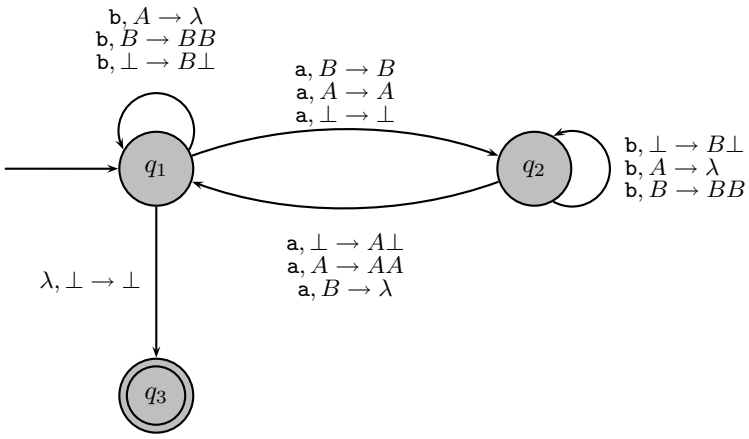
(e) dem Stackbegrenzer $\perp$

(f) dem Acceptszustand $F = \{q_3\}$

(g) und der Übergangsfunktion

$$
\begin{aligned}
(q_1, \mathtt{a}, \bot) &= \{(q_2, \bot)\} \\
(q_1, \mathtt{a}, \mathtt{A}) &= \{(q_2, \mathtt{A})\} \\
(q_1, \mathtt{a}, \mathtt{B}) &= \{(q_2, \mathtt{B})\} \\
\\
(q_1, \mathtt{b}, \bot) &= \{(q_1, \mathtt{B}\bot)\} \\
(q_1, \mathtt{b}, \mathtt{B}) &= \{(q_1, \mathtt{BB})\} \\
(q_1, \mathtt{b}, \mathtt{A}) &= \{(q_1, \lambda)\} \\
\\
(q_2, \mathtt{a}, \bot) &= \{(q_1, \mathtt{A}\bot)\} \\
(q_2, \mathtt{a}, \mathtt{A}) &= \{(q_1, \mathtt{AA})\} \\
(q_2, \mathtt{b}, \bot) &= \{(q_2, \mathtt{B}\bot)\} \\
(q_2, \mathtt{b}, \mathtt{A}) &= \{(q_2, \lambda\} \\
(q_2, \mathtt{b}, \mathtt{B}) &= \{(q_2, \mathtt{BB}\} \\
(q_2, \mathtt{a}, \mathtt{B}) &= \{(q_1, \lambda\} \\
(q_1, \lambda.\bot) &= \{(q_3, \bot)\}.
\end{aligned}
$$

Das Zustandsdiagramm dieses Automaten ist



Der String aaabba wird folgendermaßen verarbeitet:

$$\begin{aligned}(q_1, \texttt{aaabba}, \bot) &\longrightarrow (q_2, \texttt{aabba}, \bot)\\ &\longrightarrow (q_1, \texttt{abba}, A\bot)\\ &\longrightarrow (q_2, \texttt{bba}, A\bot)\\ &\longrightarrow (q_2, \texttt{ba}, \bot)\\ &\longrightarrow (q_2, \texttt{a}, B\bot)\\ &\longrightarrow (q_1, \lambda, \bot)\\ &\longrightarrow (q_3, \lambda, \bot) \quad \longrightarrow \text{accept.}\end{aligned}$$

Der String bbaaaabaa wird folgendermaßen verarbeitet:

$$\begin{aligned}(q_1, \texttt{bbaaaabaa}, \bot) &\longrightarrow (q_1, \texttt{baaaabaa}, B\bot)\\ &\longrightarrow (q_1, \texttt{aaaabaa}, BB\bot)\\ &\longrightarrow (q_2, \texttt{aaabaa}, BB\bot)\\ &\longrightarrow (q_1, \texttt{aabaa}, B\bot)\\ &\longrightarrow (q_2, \texttt{abaa}, B\bot)\\ &\longrightarrow (q_1, \texttt{baa}, \bot)\\ &\longrightarrow (q_1, \texttt{aa}, B\bot)\\ &\longrightarrow (q_2, \texttt{a}, B\bot)\\ &\longrightarrow (q_1, \lambda, \bot)\\ &\longrightarrow (q_3, \lambda, \bot) \quad \longrightarrow \text{accept.}\end{aligned}$$

Der String aabb:

$$\begin{aligned}(q_1, \texttt{aabb}, \bot) &\longrightarrow (q_2, \texttt{abb}, \bot)\\ &\longrightarrow (q_1, \texttt{bb}, A\bot)\\ &\longrightarrow (q_1, \texttt{b}, \bot)\\ &\longrightarrow (q_1, \lambda, B\bot) \quad \longrightarrow \text{not accept}\end{aligned}$$

A.4.4 Übung [4.45]

Geben Sie für jede der folgenden Sprachen über dem Alphabet $\Sigma = \{0, 1\}$ bzw. $\Sigma = \{a, b, c\}$ eine kontextfreie Grammatik G an mit $L(G) = L_i, i = 1, \ldots 7$.

(a) $L_1 = \{0^n 1^m, m \geq n, n \geq 1\}$,

(b) $L_2 = \{0^n 1^n, n \geq 1\} \cup \{0^n 1^{2n}, n \geq 1\}$

(c) $L_3 = \{0^n 1^m 0^{n+m}, m, n \geq 1\}$,

(d) $L_4 = \{0^{2n} 1^m, n \geq 1, m \geq n\}$,

(e) $L_5 = \{0^n 1^m, m, n \geq 1 m \neq n\}$,

(f) $L_6 = \{w \in \{a, b, c\}^* \mid a^i b^j c^k, i + j \geq 1, k \leq i + j\}$,

(g) $L_7 = \{w \in \{a, b, c\}^* \mid a^k b^l c^m, k, l, m \geq 1, k = 2l \text{ oder } m = 2l\}$.

Lösung:

(a) Die Sprache

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

wird durch die folgende kontextfreie Grammatik erzeugt:

$$G_1 = (N, T, P, S)$$

mit

$$N = \{S, A\}, T = \{0, 1\},$$

Startvariable S und Produktionen

$$\begin{array}{lcl} S & \longrightarrow & 0S1 \mid 01 \mid 0A1 \\ A & \longrightarrow & A1 \mid 1 \end{array}$$

(b) Die Sprache

$$L_2 = \{0^n 1^n, n \geq 1\} \cup \{0^n 1^{2n}, n \geq 1\}$$

besteht aus Wörtern wie

$$L_2 = \{01, 011, 0011, 001111, 000111, 000111111, \ldots\}.$$

Die Grammatik muss Strings erzeugen, die gleich viele 0en wie 1en haben *oder* doppelt so viele 1en wie 0en. Damit

$$G_2 = (N, T, P, S)$$

mit $N = \{S, A, B\}$, $T = \{0, 1\}$, Startvariable S und den Produktionen

$$\begin{array}{lcl} S & \longrightarrow & 0A1 \mid 01 \mid 0B11 \mid 011 \\ A & \longrightarrow & 0A1 \mid 01 \\ B & \longrightarrow & 0B11 \mid 011 \end{array}$$

(c) Die Sprache

$$L_3 = \{\mathtt{0}^n\mathtt{1}^m\mathtt{0}^{n+m}, m, n \geq 1\}$$

wird von der folgenden Grammatik G_3 erzeugt:

$$G_3 = (N, T, P, S),$$

mit $N = \{S, A\}, T = \{\mathtt{0}, \mathtt{1}\}$ Startvariable S und Produktionen

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}S\mathtt{0} \mid \mathtt{1}A\mathtt{0} \mid \mathtt{10} \\ A & \longrightarrow & \mathtt{1}A\mathtt{0} \mid \mathtt{10} \end{array}$$

(d) Die Sprache

$$L_4 = \{\mathtt{0}^{2n}\mathtt{1}^m, n \geq 1, m \geq n\}$$

enthält Stings der Form

$$L_4 = \{\underbrace{\mathtt{001}, \mathtt{0011}, \mathtt{00111}, \ldots}_{n=1}, \underbrace{\mathtt{000011}, \mathtt{0000111}, \mathtt{00001111}, \ldots\ldots}_{n=2}\}$$

Die folgende kontextfreie Grammatik erzeugt diese Sprache

$$G_4 = (N, T, P, S),$$

mit $N = \{S, A\}, T = \{\mathtt{0}, \mathtt{1}\}$ Startvariable S und Produktionen

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{00}S\mathtt{1} \mid \mathtt{00}A\mathtt{1} \mid \mathtt{001} \\ A & \longrightarrow & A\mathtt{1} \mid \mathtt{1} \mid \mathtt{001}. \end{array}$$

(e) Die Sprache

$$L_5 = \{\mathtt{0}^n\mathtt{1}^m, m, n \geq 1 m \neq n\}$$

wird durch die folgende kontextfreie Grammatik erzeugt:

$$G_5 = (N, T, P, S)$$

mit den Variablen

$$N = \{S, A, B\}$$

den Terminalzeichen $T = \{\mathtt{0}, \mathtt{1}\}$, der Startvariablen S und den Produktionen

$$\begin{array}{lcl} S & \longrightarrow & \mathtt{0}S\mathtt{1} \mid \mathtt{0}A\mathtt{1} \mid \mathtt{0}B\mathtt{1} \\ A & \longrightarrow & \mathtt{0}A \mid \mathtt{0} \\ B & \longrightarrow & B \mid \mathtt{1} \mid \mathtt{1} \end{array}$$

Die Produktion $S \to \mathtt{0}S\mathtt{1}$ erzeugt zunächst Strings mit gleich vielen 0en und 1en. Durch die Produktionen $S \to \mathtt{0}A\mathtt{1}$ oder $S \to \mathtt{0}B\mathtt{1}$ wird erreicht, dass durch die A Variable mehr 0en als 1en erzeugt werden, oder durch die B Variable mehr 1en als 0en. In jedem Fall entstehen dadurch 01Folgen der Form $\mathtt{0}^n\mathtt{1}^m$ wobei die Potenzen nicht gleich sind.

(f) Die Sprache

$$L_6 = \{w \in \{a,b,c\}^* \mid a^i b^j c^k, i+j \geq 1, k \leq i+j\}.$$

Diese

(g) Die Sprache

$$L_7 = \{w \in \{a,b,c\}^* \mid a^k b^l c^m, k,l,m \geq 1, k = 2l \text{ oder } m = 2l\}.$$

enthält Worte der Form

$$aab, aaaabbb, aaaaaabbb, \ldots$$

oder

$$bcc, bbcccc, bbbccccccc, \ldots$$

Diese Sprache wird generiert durch

$$G_7 = (N,T,P,S)$$

mit $N = \{S,A,B\}, T = \{a,b,c\}$, Startvariable S und den Produktionen

$$\begin{array}{lcl} S & \longrightarrow & aaAb \mid aab \\ S & \longrightarrow & bBcc \mid bcc \\ A & \longrightarrow & aaAb \mid aab \\ B & \longrightarrow & bBcc \mid bcc \end{array}$$

A.4.5 Übung [4.46]

Gegeben ist die folgende Sprache

$$L = \{w \in \{a,b\}^* \mid w = (a^2b)^n, n \geq 0, n \in \mathbb{N}\}.$$

(a) Geben Sie zu dieser Sprache einen Kellerautomaten P an mit $L = L(P)$.

(b) Geben Sie zu dieser Sprache einen deterministischen endlichen Automaten M an mit $L = L(M)$.

(c) Zu welcher/welchen CHOMSKY Sprachklasse/n gehört die Sprache L?

Lösung:

Die Sprache L besteht aus Wörten der Form $aabaabaab\ldots$.

(a) Ein Kellerautomat, der die Sprache

$$L = \{w \in \{a,b\}^* \mid w = (a^2b)^n, n \geq 0, n \in \mathbb{N}\}$$

akzeptiert, arbeitet informell wie folgt, wenn ein zu akzeptierender String der Form $aabaab$ gelesen wird:
Es wird im Startzustand q_0 ein a auf dem Inputband gelesen, der Stackinhalt ist der Begrenzer $\perp$. Gleichzeitig verzweigt der Automat nichtdeterministsich in einen Acceptzustand.[4] Der Automat bleibt im Startzustand, setzt einen Merker A auf den Stack, und geht ein Zeichen weiter auf dem Inputband. Dann liest er im Zustand q_0 das zweite a und das oberste Stacksymbol ist ein A. Der Automat geht in einen Zustand q_1 über, lässt den Merker A auf dem Stack. Dann geht er zum nächsten Symbol.

- Ist das Symbol ein a, dann gibt es keinen Folgezustand, das Wort hat dann den Präfix aaa,
- Ist das Symbol ein b, dann geht der Automat in einen Acceptzustand q_3 über und nimmt den Merker A vom Stack; der Stack hat nur noch den Begrenzer $\perp$ als Inhalt.

Der Pushdown Automat ist

$$P = (Q, \Sigma, \Gamma, \delta, q^s, \perp, F)$$

mit:

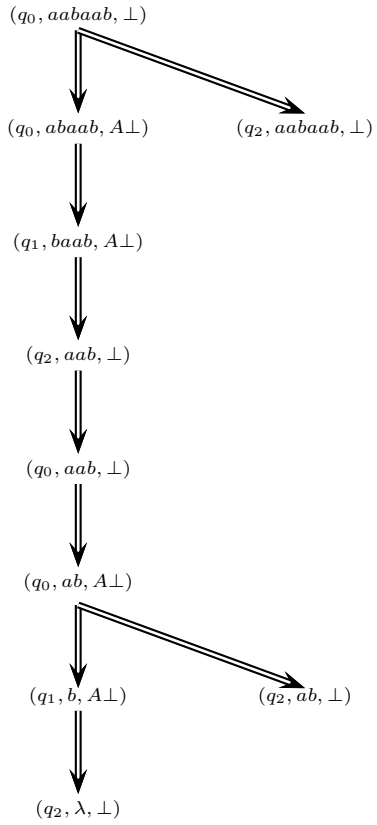
$$Q = \{q_0, q_1, q_2\}, \Sigma = \{a,b\}, \Gamma = \{A, \perp\}, q^s = q_0, F = \{q_2\}$$

und den Übergängen δ:

$$\begin{array}{lcl} (q_0, a, \perp) & \longrightarrow & (q_0, A\perp) \\ (q_0, \lambda, \perp) & \longrightarrow & (q_2, \perp) \\ (q_0, a, A) & \longrightarrow & (q_1, A) \\ (q_1, b, A) & \longrightarrow & (q_2, \lambda) \\ (q_2, a, \perp) & \longrightarrow & (q_0, A\perp). \end{array}$$

[4] Der leere String λ ist Element der Sprache.

Betrachte den String $w = aabaab \in L$. Der Automat P durchläuft bei der verarbeitung dieses Strings die folgenden Konfigurationen:



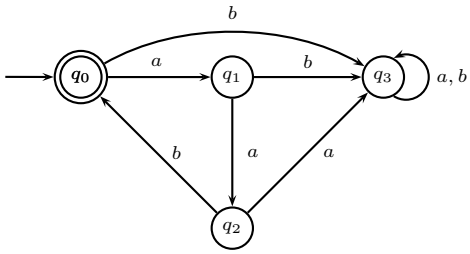
(b) Die Sprache L ist regulär, da wir einen deterministischen endlichen Automaten M angeben können, der diese Sprache akzeptiert. Dieser ist gegeben durch das 5–Tupel:

$$M = (Q, \Sigma, \delta, q^s, F),$$

mit $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $q^s = q_0$, $F = \{q_0\}$.
Die Übergangsfunktion δ ist:

	a	b
q_0	q_1	q_3
q_1	q_2	q_3
q_2	q_3	q_0
q_3	q_3	q_3

Das Zustandsdiagramm ist:



(c) Die Sprache L ist eine Typ–3 Sprache, *i.e.* eine reguläre Sprache, da ein DFA existiert, der die Sprache akzeptiert. Da Typ–3 Sprachen eine Teilmenge der Typ–2 Sprachen ist, ist L auch eine Typ–2 Sprache, usw.

A.4.6 Übung [4.47]

Betrachte die Sprache

$$L = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w \text{ enthält doppelt soviele } \mathtt{0}\text{en wie } \mathtt{1}\text{en}\}.$$

(a) Geben Sie einen Kellerautomaten an, der L akzeptiert.

(b) Zeigen Sie, wie der Automat das Wort

$$w = \mathtt{100011100000}$$

erkennt.

Lösung:

Wörter, die zu dieser Sprache gehöhern, haben doppelt soviele `0`en wie `1`en, die Reihenfolge der Zeichen ist irrelevant. Dem Kellerautomaten liegt die folgende Idee zugrunde. die Zustände haben die folgende Bedeutung:

q_0 Verhältnis der Anzahl der `0`en zu Anzahl der `1`en ist 2:1.

q_1 Verhältnis der Anzahl der `0`en zu Anzahl der `1`en ist größer als 2:1, *i.e.* zu viele `0`en.

q_2 Übergangszustand von q_1 zu q_3.

q_3 Verhältnis der Anzahl der `0`en zu Anzahl der `1`en ist kleiner als 2:1, *i.e.* zu wenig `0`en.

Der Automat ist

$$P = (Q, \Sigma, \Gamma, \delta, q^s, \bot, F)$$

mit

$$Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{\mathtt{0},\mathtt{1}\}, \Gamma = \{A, B, C, \bot\}, q^s = q_0, F = \{q_0\},$$

Stackbegrenzer ist $\bot$ und die Übergänge δ sind (die Nummerierung dient der Kennzeichnung der Übergänge):

1	$(q_0, \mathtt{0}, \bot)$	$\longrightarrow$	$(q_1, A\bot)$	setze Merker A auf den Stack
2	$(q_0, \mathtt{1}, \bot)$	$\longrightarrow$	$(q_3, B\bot)$	setze Merker B auf den Stack
3	$(q_1, \mathtt{0}, A)$	$\longrightarrow$	(q_1, AA)	
4	$(q_1, \mathtt{1}, A)$	$\longrightarrow$	(q_2, λ)	lösche Stackzeichen
5	(q_2, λ, A)	$\longrightarrow$	(q_1, λ)	lösche noch ein Stackzeichen spontan
6	$(q_1, \lambda, \bot)$	$\longrightarrow$	$(q_0, \bot)$	Endzustand erreicht
7	$(q_2, \lambda, \bot)$	$\longrightarrow$	(q_3, C)	
8	$(q_3, \mathtt{1}, B)$	$\longrightarrow$	(q_2, BB)	
9	$(q_3, \mathtt{0}, B)$	$\longrightarrow$	(q_3, C)	
10	$(q_3, \mathtt{0}, C)$	$\longrightarrow$	(q_3, λ)	
11	$(q_3, \mathtt{1}, C)$	$\longrightarrow$	(q_3, CB)	
12	$(q_3, \lambda, \bot)$	$\longrightarrow$	$(q_0, \bot)$	

Die Übergänge haben die folgende Bedeutung:

(1) Der Automat ist im Startzustand q_0, liest das erste Zeichen des Strings die `0` und der Stack ist leer. Der Automat setzt einen Merker (Zähler) A auf den Stack, dass die `0` gelesen wurde, und geht in den Zustand q_1 über, er hat mehr `0`en als `1`en gelesen.

(2) Der Automat ist im Startzustand q_0, liest als erstes Zeichen des Strings die `1`, der Stack ist leer. Der Automat setzt einen Merker (Zähler) B auf den Stack, dass die `1` gelesen wurde, und geht in den Zustand q_3 über, er hat mehr `1`en als `03`n gelesen.

Sehen wir uns an, wie die Maschine verschiedene Strings verarbeitet. Betrachte zunächst den String $w = \texttt{100010}$.

$$\begin{aligned}
(q_0\texttt{100010}, \bot) &\overset{(2)}{\Longrightarrow} (q_3\texttt{00010}, B\bot)\\
&\overset{(9)}{\Longrightarrow} (q_3\texttt{0010}, C\bot)\\
&\overset{(10)}{\Longrightarrow} (q_3\texttt{010}, \bot)\\
&\overset{(12)}{\Longrightarrow} (q_0\texttt{010}, \bot)\\
&\overset{(1)}{\Longrightarrow} (q_1\texttt{10}, A\bot)\\
&\overset{(4)}{\Longrightarrow} (q_2\texttt{0}, \bot)\\
&\overset{(7)}{\Longrightarrow} (q_3\texttt{0}, C\bot)\\
&\overset{(10)}{\Longrightarrow} (q_3\lambda, \bot)\\
&\overset{(12)}{\Longrightarrow} (q_0, \bot)
\end{aligned}$$

Betrachte den String $w = \texttt{000011}$:

$$\begin{aligned}
(q_0\texttt{000011}, \bot) &\overset{(1)}{\Longrightarrow} (q_1\texttt{00011}, A\bot)\\
&\overset{(3)}{\Longrightarrow} (q_1\texttt{0011}, AA\bot)\\
&\overset{(3)}{\Longrightarrow} (q_1\texttt{011}, AAA\bot)\\
&\overset{(3)}{\Longrightarrow} (q_1\texttt{11}, AAAA\bot)\\
&\overset{(4)}{\Longrightarrow} (q_2\texttt{1}, AAA\bot)\\
&\overset{(5)}{\Longrightarrow} (q_1\texttt{1}, AA\bot)\\
&\overset{(4)}{\Longrightarrow} (q_2\lambda, A\bot)\\
&\overset{(5)}{\Longrightarrow} (q_1\lambda, \bot)\\
&\overset{(5)}{\Longrightarrow} (q_0, \bot)
\end{aligned}$$

Betrachte den String $w = \texttt{100011100000}$:

$$
\begin{aligned}
(q_0\texttt{100011100000}, \bot) &\overset{(2)}{\Longrightarrow} (q_3\texttt{00011100000}, B\bot)\\
&\overset{(9)}{\Longrightarrow} (q_3\texttt{0011100000}, C\bot)\\
&\overset{(10)}{\Longrightarrow} (q_3\texttt{011100000}, \bot)\\
&\overset{(12)}{\Longrightarrow} (q_0\texttt{011100000}, \bot)\\
&\overset{(1)}{\Longrightarrow} (q_1\texttt{11100000}, A\bot)\\
&\overset{(4)}{\Longrightarrow} (q_2\texttt{1100000}, \bot)\\
&\overset{(7)}{\Longrightarrow} (q_3\texttt{1100000}, C\bot)\\
&\overset{(11)}{\Longrightarrow} (q_3\texttt{100000}, CB\bot)\\
&\overset{(11)}{\Longrightarrow} (q_3\texttt{00000}, CBB\bot)\\
&\overset{(10)}{\Longrightarrow} (q_3\texttt{0000}, BB\bot)\\
&\overset{(9)}{\Longrightarrow} (q_3\texttt{000}, CB\bot)\\
&\overset{(10)}{\Longrightarrow} (q_3\texttt{00}, B\bot)\\
&\overset{(9)}{\Longrightarrow} (q_3\texttt{0}, C\bot)\\
&\overset{(10)}{\Longrightarrow} (q_3\lambda, \bot)\\
&\overset{(12)}{\Longrightarrow} (q_0, \bot)
\end{aligned}
$$

A.4.7 Übung [4.48]

Gegeben ist die kontextfreie Grammatik

$$G = (N, T, P, S)$$

mit

$$\begin{aligned} N &= \{S\} \\ T &= \{a, b\} \\ P &= \{S \longrightarrow aab \mid bbS\}. \end{aligned}$$

(a) Geben Sie einen Pushdown Automaten

$$M = (Q, \Sigma, \Gamma, \delta, q^s, \bot)$$

an, der zur Grammatik G äquivalent ist.

(b) Geben Sie die Konfigurationsübergänge von M für das Testwort $w = bbbbaab$ an.

Lösung:

Wir wenden das Verfahren aus Abschnitt [4.7] an, um aus einer gegebenen kontextfreien Grammatik einen Pushdown Automaten zu konstruieren. Damit ist

$$M = (Q, \Sigma, \Gamma, \delta, q^s, \bot)$$

mit der Menge der Zustände

$$Q = \{q_1, q_2, q_3\},$$

dem Inputalphabet

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

dem Stackalphabet

$$\Gamma = \{S, a, b, \bot\},$$

dem Startzustand

$$q^s = q_1,$$

dem Stackbegrenzer $\bot$, dem Acceptzustand $F = \{q_3\}$ und den Übergängen

$$\delta(q_1, \lambda, \bot) = (q_2, S\bot), \tag{A.2a}$$
$$\delta(q_2, \lambda, S) = (q_2, bbS), \tag{A.2b}$$
$$\delta(q_2, \lambda, S) = (q_2, aab), \tag{A.2c}$$
$$\delta(q_2, a, a) = (q_2, \lambda), \tag{A.2d}$$
$$\delta(q_2, b, b) = (q_2, \lambda), \tag{A.2e}$$
$$\delta(q_2, \lambda, \bot) = (q_3, \bot). \tag{A.2f}$$

Wir betrachten die Verarbeitung des Testwortes $w = bbbbaab$ mit diesem Automaten:

$$\begin{aligned}
(q_1 bbbbaab, \bot) &\overset{(A.2a)}{\Longrightarrow} (q_2 bbbbaab, S\bot)\\
&\overset{(A.2b)}{\Longrightarrow} (q_2 bbbbaab, bbS\bot)\\
&\overset{(A.2e)}{\Longrightarrow} (q_2 bbbaab, bS\bot)\\
&\overset{(A.2e)}{\Longrightarrow} (q_2 bbaab, S\bot)\\
&\overset{(A.2b)}{\Longrightarrow} (q_2 bbaab, bbS\bot)\\
&\overset{(A.2e)}{\Longrightarrow} (q_2 baab, bS\bot)\\
&\overset{(A.2e)}{\Longrightarrow} (q_2 aab, S\bot)\\
&\overset{(A.2c)}{\Longrightarrow} (q_2 aab, aab\bot)\\
&\overset{(A.2d)}{\Longrightarrow} (q_2 ab, ab\bot)\\
&\overset{(A.2d)}{\Longrightarrow} (q_2 b, b\bot)\\
&\overset{(A.2e)}{\Longrightarrow} (q_2 \lambda, \bot)\\
&\overset{(A.2f)}{\Longrightarrow} (q_3, \bot).
\end{aligned}$$

Alternativ kann der folgende Automat angegeben werden:

Die von der gegebenen Grammatik generierte Sprache ist

$$L = \{w \in \{a,b\}^* \mid w = b^{2n}aab,\ n \in \mathbb{N}\}.$$

Der folgende Pushdown Automat akzeptiert ebenfalls alle Worte dieser Form:

$$M = (Q, \Sigma, \Gamma, \delta, q^2, \bot, F)$$

mit

$$Q = \{q_0, q_1, q_2, q_3, q_4\}, \Sigma = \{a,b\}, \Gamma = \{\bot\}, q^s = q_0, \bot, F = \{q_4\}$$

und den Übergängen:

$$\begin{array}{c||lcl}
1 & (q_0, b, \bot) & \longrightarrow & (q_1, \bot)\\
2 & (q_1, b, \bot) & \longrightarrow & (q_0, \bot)\\
3 & (q_0, a, \bot) & \longrightarrow & (q_2, \bot)\\
4 & (q_2, a, \bot) & \longrightarrow & (q_3, \bot)\\
5 & (q_3, b, \bot) & \longrightarrow & (q_4, \bot)
\end{array}$$

Dieser Automat macht keinen Gebrauch von dem Stack, dies impliziert, dass es ein endlicher Automat gibt, der die Sprache erkennt.

Dieser Automat verarbeitet den String $w = bbbbaab$ wie folgt;

$$\begin{aligned}
(q_0 bbbbaab, \bot) &\xrightarrow{(1)} (q_1 bbbaab, \bot) \\
&\xrightarrow{(2)} (q_0 bbaab, \bot) \\
&\xrightarrow{(1)} (q_1 baab, \bot) \\
&\xrightarrow{(2)} (q_0 aab, \bot) \\
&\xrightarrow{(3)} (q_2 ab, \bot) \\
&\xrightarrow{(4)} (q_3 b, \bot) \\
&\xrightarrow{(5)} (q_4, \bot)
\end{aligned}$$

A.5 Lösungen zu den Übungen aus Kapitel [5.6]

A.5.1 Übung [5.49]

Gegeben ist die folgende TURING Maschine

$$M = (Q, \Sigma, \delta, q^s, q^f)$$

mit

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\} \\ \Sigma &= \{\ |\ \}, \\ q^s &= q_0, \\ q^f &= q_8 \end{aligned}$$

und den Übergängen

$$\begin{array}{lcl} q_0\ | & \longmapsto & q_1\ \sqcup \\ q_1\ \sqcup & \longmapsto & q_2\ R \\ q_2\ | & \longmapsto & q_3\ \sqcup \\ q_3\ \sqcup & \longmapsto & q_4\ R \\ q_4\ | & \longmapsto & q_5\ \sqcup \\ q_5\ \sqcup & \longmapsto & q_0\ R \\ q_0\ \sqcup & \longmapsto & q_6\ | \\ q_6\ | & \longmapsto & q_6\ R \\ q_6\ \sqcup & \longmapsto & q_7\ | \\ q_7\ | & \longmapsto & q_7\ R \\ q_7\ \sqcup & \longmapsto & q_8\ | \\ q_2\ \sqcup & \longmapsto & q_8\ | \\ q_4\ \sqcup & \longmapsto & q_7\ | \end{array}$$

Diese TURING Maschine berechnet eine Funktion $f(n)$, wobei n eine natürliche Zahl in unärer Darstellung ist. Welche Funktion berechnet diese Maschine?

Lösung:

Um zu sehen, welche Funktion die vorgegebene TURING Maschine berechnet, sieht man sich am einfachsten an, wie die Maschine einen String verarbeitet. Als Beispiel sehen wir uns an, wie der String

$$w = | \ | \ | \ | \ |$$

verarbeitet wird, der die natürliche Zahl 4 codiert. Die Schritte der Verarbeitung sind in der Abbildung [A.26] aufgeführt.

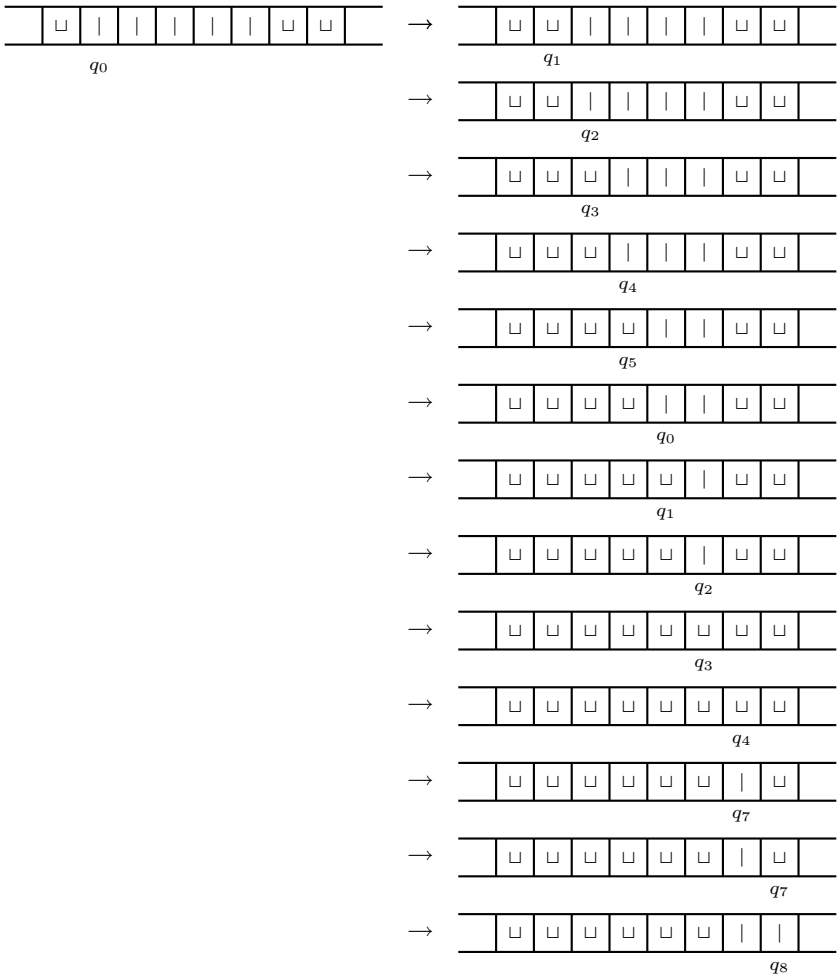


Abbildung A.26: Verarbeitung

Das Ergebnis ist also die natürliche Zahl 1. Die Maschine hat also berechnet:

$$f(4) = 1.$$

Analog kann man zeigen, dass diese `TM` folgende Werte berechnet:

$$f(1) = 1, f(2) = 2, f(3) = 0, f(4) = 1, f(5) = 2, f(6) = 0 \qquad \text{usw.},$$

Mit anderen Worten, die obige TURING Maschine berechnet die Funktion

$$f(n) = n \bmod 3.$$

A.5.2 Übung [5.50]

Man gebe eine TURING Maschine an, die die Funktion

$$f(n) = 2n$$

berechnet, wobei n eine natürliche Zahl in unärer Darstellung ist.

Lösung:

Informell arbeitet die Maschine wie folgt:

step 1: Das erste | Zeichen wird gelesen, und gelöscht.

step 2: Der Schreib–/Lesekopf wandert nach rechts, liest die | bis zum ersten Blank hinter dem Eingabewort.

step 3: Der Automat lässt das Blank stehen, geht eine Zelle nach rechts.

step 4: Der Automat schreibt in die Zelle ein |.

step 5: Der Automat geht eine Zelle nach rechts.

step 6: Der Automat schreibt in die Zelle das zweite |.

step 7: Der Automat geht nach links bis zum ersten Blank vor dem Inputstring.

step 8: Dann geht er eine Zelle nach rechts und liest ein |–Zeichen.

Der Automat liest jedes Inputzeichen, löscht es, und schreibt für jedes gelöschte | zwei Striche hinter den Inputstring auf das Band, dadurch wird die Anzahl der Striche verdoppelt. Damit durch dieses Verfahren das korrekte Ergebnis auf dem Band steht, muss zum Schluß noch ein Strich gelöscht werden, denn für die Zahl $n \in \mathbb{N}$ werden $n + 1$ Striche auf das Band geschrieben. Wird jeder Strich verdoppelt, stehen am Schluß $2n + 2$ Striche auf dem Band. Die Zahl $2n \in \mathbb{N}$ wird aber durch $2n + 1$ Striche im unären Alphabet dargestellt.

Die Ausgangskonfiguration ist: Der Inputstring mit $n+1$ Strichen steht auf dem Eingabeband, der Schreib–/Lesekopf steht auf der Zelle mit dem ersten |, der

Automat ist im Startzustand q_0. Die Übergangsfunktion ist:

(1)	q_0 |	$\longmapsto$	q_1 ⊔	Löschen des ersten |
(2)	q_1 ⊔	$\longmapsto$	q_2 R	Gehe eine Zelle nach rechts
(3)	q_2 ⊔	$\longmapsto$	q_3 R	
(4)	q_2 |	$\longmapsto$	q_2 R	Geh eine Zelle nach rechts
(5)	q_3 ⊔	$\longmapsto$	q_4 |	Schreib das zweite |
(6)	q_3 |	$\longmapsto$	q_3 R	
(7)	q_4 ⊔	$\longmapsto$	q_5 |	Schreib das zweite |
(8)	q_4 |	$\longmapsto$	q_4 R	
(9)	q_5 |	$\longmapsto$	q_6 L	
(10)	q_6 ⊔	$\longmapsto$	q_7 L	
(11)	q_6 |	$\longmapsto$	q_6 L	
(12)	q_7 |	$\longmapsto$	q_8 L	Scan der Input | nach links
(13)	q_7 ⊔	$\longmapsto$	q_9 R	Anfang erreicht, geh nach rechts
(14)	q_8 |	$\longmapsto$	q_8 L	Lösche ein Strich, dann sind noch $2n+1$ Striche
(15)	q_8 ⊔	$\longmapsto$	q_0 R	Anfang erreicht, geh nach rechts
(16)	q_9 ⊔	$\longmapsto$	q_9 R	Anfang erreicht, geh nach rechts
(17)	q_9 |	$\longmapsto$	q_E ⊔	Anfang erreicht, geh nach rechts

Wir sehen uns die Verarbeitung des Strings ||| an, der die Zahl 2 repräsentiert.

$$
\begin{aligned}
⊔q_0\,|||\,⊔ &\xrightarrow{(1)} ⊔q_1⊔\,||\,⊔\\
&\xrightarrow{(2)} ⊔\,⊔\,q_2\,||\,⊔\\
&\xrightarrow{(4)} ⊔⊔\,|\,q_2\,|\,⊔\\
&\xrightarrow{(4)} ⊔⊔\,||\,q_2⊔\\
&\xrightarrow{(3)} ⊔⊔\,||\,⊔q_3⊔\\
&\xrightarrow{(5)} ⊔⊔\,||\,⊔q_4\,|\\
&\xrightarrow{(8)} ⊔⊔\,||\,⊔\,|\,q_4⊔\\
&\xrightarrow{(7)} ⊔⊔\,||\,⊔\,|\,q_5\,|\\
&\xrightarrow{(9)} ⊔⊔\,||\,⊔q_6\,||\\
&\xrightarrow{(11)} ⊔⊔\,||\,q_6⊔\,||\\
&\xrightarrow{(10)} ⊔⊔\,|\,q_7\,|\,⊔\,||\\
&\xrightarrow{(12)} ⊔\,⊔\,q_8\,||\,⊔\,||\\
&\xrightarrow{(14)} ⊔q_8⊔\,||\,⊔\,||\\
&\xrightarrow{(15)} ⊔\,⊔\,q_0\,||\,⊔\,||
\end{aligned}
$$

$$
\begin{array}{ll}
\xrightarrow{(1)} & \sqcup\sqcup q_1\sqcup \,||\, \sqcup \,||\\
\xrightarrow{(2)} & \sqcup\sqcup\sqcup q_2 \,|\, \sqcup \,||\, \sqcup\\
\xrightarrow{(4)} & \sqcup\sqcup \,|\, q_2\sqcup \,||\, \sqcup\\
\xrightarrow{(3)} & \sqcup\sqcup \,|\, \sqcup q_3 \,||\, \sqcup\\
\xrightarrow{(6)} & \sqcup\sqcup \,|\, \sqcup \,|\, q_3 \,|\, \sqcup\\
\xrightarrow{(6)} & \sqcup\sqcup \,|\, \sqcup \,||\, q_3\sqcup\\
\xrightarrow{(5)} & \sqcup\sqcup \,|\, \sqcup \,||\, q_4 \,|\, \sqcup\\
\xrightarrow{(8)} & \sqcup\sqcup \,|\, \sqcup \,|||\, q_4\sqcup\\
\xrightarrow{(7)} & \sqcup\sqcup \,|\, \sqcup \,|||\, q_5 \,|\, \sqcup\\
\xrightarrow{(9)} & \sqcup\sqcup \,|\, \sqcup \,||\, q_6 \,||\, \sqcup\\
\xrightarrow{(11)} & \sqcup\sqcup \,|\, \sqcup \,|\, q_6 \,|||\, \sqcup\\
\xrightarrow{(11)} & \sqcup\sqcup \,|\, \sqcup q_6 \,||||\, \sqcup\\
\xrightarrow{(11)} & \sqcup\sqcup \,|\, q_6\sqcup \,||||\, \sqcup\\
\xrightarrow{(10)} & \sqcup\sqcup q_7 \,|\, \sqcup \,||||\, \sqcup\\
\xrightarrow{(12)} & \sqcup q_8\sqcup \,|\, \sqcup \,||||\, \sqcup\\
\xrightarrow{(15)} & \sqcup\sqcup q_0 \,|\, \sqcup \,||||\, \sqcup\\
\xrightarrow{(1)} & \sqcup\sqcup q_1\sqcup\sqcup \,||||\, \sqcup\\
\xrightarrow{(3)} & \sqcup\sqcup\sqcup\sqcup q_3 \,||||\, \sqcup\\
\xrightarrow{(6)} & \sqcup\sqcup\sqcup\sqcup \,|\, q_3 \,|||\, \sqcup\\
\xrightarrow{(6)} & \sqcup\sqcup\sqcup\sqcup \,||\, q_3 \,||\, \sqcup\\
\xrightarrow{(6)} & \sqcup\sqcup\sqcup\sqcup \,|||\, q_3 \,|\, \sqcup\\
\xrightarrow{(6)} & \sqcup\sqcup\sqcup\sqcup \,||||\, q_3\sqcup\\
\xrightarrow{(5)} & \sqcup\sqcup\sqcup\sqcup \,||||\, q_4 \,|\, \sqcup\\
\xrightarrow{(8)} & \sqcup\sqcup\sqcup\sqcup \,|||||\, q_4\sqcup\\
\xrightarrow{(7)} & \sqcup\sqcup\sqcup\sqcup \,|||||\, q_5 \,|\, \sqcup\\
\xrightarrow{(9)} & \sqcup\sqcup\sqcup\sqcup \,||||\, q_6 \,||\, \sqcup\\
\vdots & \vdots\\
\xrightarrow{(11)} & \sqcup\sqcup\sqcup\sqcup q_6 \,||||||\, \sqcup\\
\xrightarrow{(11)} & \sqcup\sqcup\sqcup q_6\sqcup \,||||||\, \sqcup\\
\xrightarrow{(10)} & \sqcup\sqcup q_7\sqcup\sqcup \,||||||\, \sqcup\\
\xrightarrow{(13)} & \sqcup\sqcup\sqcup q_9\sqcup \,||||||\, \sqcup\\
\xrightarrow{(16)} & \sqcup\sqcup\sqcup\sqcup q_9 \,||||||\, \sqcup\\
\xrightarrow{(17)} & \sqcup\sqcup\sqcup\sqcup q_E\sqcup \,|||||\, \sqcup
\end{array}
$$

A.5.3 Übung [5.51]

Geben Sie eine TURING Maschine an, die für natürliche Zahlen in Binärdarstellung die Nachfolgerfunktion $f(n) = n + 1$ berechnet.

Hinweis: Man überlegt sich anhand einfacher Beispiele, wie man die Nachfolger erhält. Dazu ist es zweckmäßig, Beispiele von geraden und ungeraden Zahlen zu betrachten, *e.g.* `100110` oder `11001`.

Lösung:

Informell können wir die Verarbeitung der `TM` folgendermaßen beschreiben:

Gerade Zahl Betrachte den Inputstring `100110`. Um hier den Nachfolger zu berechnen, geht die `TM` zu dem kleinstmöglichen Bit ganz rechts und ersetzt die `0` durch die `1`.

Ungerade Zahl Betrachte den Inputstring `10011`. Um hier den Nachfolger zu berechnen, geht die `TM` zu dem kleinstmöglichen Bit ganz rechts. Die `1` wird gelesen, diese wird durch `0` ersetzt. Dadurch entsteht ein Übertrag in die nächste Stelle, dies muss sich die `TM` merken, dazu geht sie in einen anderen Zzustand über. Dann geht sie eine Stelle nach links und liest das Zeichen.

- Ist dieses Zeichen eine `0`, ersetze diese `0` durch `1` $\Longrightarrow$ Stop.
- Ist dieses Zeichen eine `1`, ersetze diese `1` durch eine `0` und bleibe im Übertragzustand. Gehe solange nach links und ersetze `1`en durch `0`, bis eine `0` gelesen wird. Dann ersetze diese durch `1` $\Longrightarrow$ Stop
- Wenn hinter der letzten gelesenen `1` kein Zeichen mehr folgt, schreibe in die leere Zelle vor dem Input eine `1` $\Longrightarrow$ Stop.

Die TURINGS Maschine M, die dieses Verfahren umsetzt ist ein Quintupel

$$M = (Q, \Sigma, \delta, q^s, q_e),$$

mit den vier Zuständen

$$Q = \{q_0, q_1, q_2, q_e\},$$

dem Startzustand $q^s = q_0$, dem Acceptzustand q_e, dem Inputalphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$ und den folgenden Übergängen

(1)	$q_0\texttt{1}$	$\longmapsto$	q_0R	nach rechts bis zum $\sqcup$ hinter dem Input
(2)	$q_0\texttt{0}$	$\longmapsto$	q_0R	nach rechts bis zum $\sqcup$ hinter dem Input
(3)	$q_0\sqcup$	$\longmapsto$	q_1L	Ende erreicht, eine Zelle nach links
(4)	$q_1\texttt{0}$	$\longmapsto$	$q_e\texttt{1}$	Letzte Stelle `0`, ersetze durch `1` und terminiere
(5)	$q_1\texttt{1}$	$\longmapsto$	$q_2\texttt{0}$	Letzte Stelle `1`, ersetze durch `0` und merken
(6)	$q_2\texttt{0}$	$\longmapsto$	q_1L	eine Zelle nach links
(7)	$q_1\sqcup$	$\longmapsto$	$q_e\texttt{1}$	wenn Blank gelesen wird, schreibe eine `1` und terminiere.

Sehen wir uns die Verarbeitung des String 10111 an.

$$
\begin{aligned}
q_0\texttt{101111} &\xrightarrow{(1)} 1q_0\texttt{01111}\sqcup\\
&\xrightarrow{(2)} \texttt{10}q_0\texttt{1111}\sqcup\\
&\xrightarrow{(1)} \texttt{101}q_0\texttt{111}\sqcup\\
&\xrightarrow{(1)} \texttt{1011}q_0\texttt{11}\sqcup\\
&\xrightarrow{(1)} \texttt{10111}q_0\texttt{1}\sqcup\\
&\xrightarrow{(1)} \texttt{101111}q_0\sqcup\\
&\xrightarrow{(3)} \texttt{10111}q_1\texttt{1}\sqcup\\
&\xrightarrow{(5)} \texttt{10111}q_2\texttt{0}\sqcup\\
&\xrightarrow{(6)} \texttt{1011}q_1\texttt{10}\sqcup\\
&\xrightarrow{(5)} \texttt{1011}q_2\texttt{00}\sqcup\\
&\xrightarrow{(6)} \texttt{101}q_1\texttt{100}\sqcup\\
&\xrightarrow{(5)} \texttt{101}q_2\texttt{000}\sqcup\\
&\xrightarrow{(6)} \texttt{10}q_1\texttt{1000}\sqcup\\
&\xrightarrow{(5)} \texttt{10}q_2\texttt{0000}\sqcup\\
&\xrightarrow{(6)} 1q_1\texttt{0000}\sqcup\\
&\xrightarrow{(4)} 1q_e\texttt{1000}\sqcup
\end{aligned}
$$

Output: 11000.

A.5.4 Übung [5.52]

Es gibt TURING Maschinen, die, angesetzt auf das leere Eingabeband, eine endliche Folge von Strichen auf das Band schreiben und anschließend im Endzustand terminieren. Solche TURING Maschinen heißen **Busy–Beaver Turing Maschinen**, denn man kann sich die Striche als Holzstämme vorstellen, die ein Biber für seinen Dammbau heranschleppt.[5]

Gegeben ist die folgende TURING Maschine

$$M = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}, \{\ |\ \}, \delta, q_0, q_7)$$

mit den Übergängen δ:

$$\begin{array}{llll} q_0\ \sqcup & \longmapsto & q_3\ | & \\ q_0\ | & \longmapsto & q_2\ L & \\ q_1\ \sqcup & \longmapsto & q_4\ | & \\ q_1\ | & \longmapsto & q_7\ | & \\ q_2\ \sqcup & \longmapsto & q_5\ | & \end{array} \qquad \begin{array}{llll} q_2\ | & \longmapsto & q_6\ \sqcup & \\ q_3\ | & \longmapsto & q_1\ R & \\ q_4\ | & \longmapsto & q_2\ R & \\ q_5\ | & \longmapsto & q_0\ L & \\ q_6\ \sqcup & \longmapsto & q_1\ L & \end{array}$$

Beschreiben Sie die Arbeitsweise dieser Maschine, wenn sie auf das leere Band angesetzt ist. Wieviele Stämme schleppt der fleißige Biber zu seinem Damm?

Lösung:

Eine Biber–TURING Maschine ist eine einfache `TM`, die pro Schritt genau eine 'Baumaktion' ausführt. Das heisst, erlaubt sind die folgenden Aktionen in einem Schritt:

- Sie schreibt in eine Zelle einen Strich |.
- Sie löscht in einer Zelle ein | Zeichen.
- Sie geht nach rechts.
- Sie geht nach links.

Insbesondere darf die Biber–TURING Maschine nicht auf einer Eingabezelle verharren, oder über die Zellen laufen. Die Übergangsrelation ist:

$$\begin{array}{lllll} (1) & q_0\ \sqcup & \longmapsto & q_3\ | & \\ (2) & q_0\ | & \longmapsto & q_2\ L & \\ (3) & q_1\ \sqcup & \longmapsto & q_4\ | & \\ (4) & q_1\ | & \longmapsto & q_7\ | & \\ (5) & q_2\ \sqcup & \longmapsto & q_5\ | & \\ (6) & q_2\ | & \longmapsto & q_6\ \sqcup & \\ (7) & q_3\ | & \longmapsto & q_1\ R & \\ (8) & q_4\ | & \longmapsto & q_2\ R & \\ (9) & q_5\ | & \longmapsto & q_0\ L & \\ (10) & q_6\ \sqcup & \longmapsto & q_1\ L & \end{array}$$

[5] Weitere Informationen über dieses Thema findet man in [8], Kapitel 39.

Betrachte ein leeres Eingabeband. Die oben skizzierte Maschine arbeitet folgendermaßen:

$$
\begin{aligned}
\cdots \sqcup \sqcup q_0 \sqcup \sqcup \sqcup \cdots &\xrightarrow{(1)} \sqcup q_3 \mid \sqcup\sqcup \\
&\xrightarrow{(7)} \sqcup \mid q_1 \sqcup \sqcup \\
&\xrightarrow{(3)} \sqcup \mid q_4 \mid \sqcup\sqcup \\
&\xrightarrow{(8)} \sqcup \mid\mid q_2 \sqcup \sqcup \\
&\xrightarrow{(5)} \sqcup \mid\mid q_5 \mid \sqcup \\
&\xrightarrow{(9)} \sqcup \mid q_0 \mid\mid \sqcup \\
&\xrightarrow{(2)} \sqcup q_2 \mid\mid\mid \sqcup \\
&\xrightarrow{(6)} \sqcup q_6 \sqcup \mid\mid \sqcup \\
&\xrightarrow{(10)} \sqcup q_1 \sqcup \sqcup \mid\mid \sqcup \\
&\xrightarrow{(3)} \sqcup q_4 \mid \sqcup \mid\mid \sqcup \\
&\xrightarrow{(8)} \sqcup \mid q_2 \sqcup \mid\mid \sqcup \\
&\xrightarrow{(5)} \sqcup \mid q_5 \mid\mid\mid \sqcup \\
&\xrightarrow{(9)} \sqcup q_0 \mid\mid\mid\mid \sqcup \\
&\xrightarrow{(2)} \sqcup q_2 \sqcup \mid\mid\mid\mid \sqcup \\
&\xrightarrow{(5)} \sqcup q_5 \mid\mid\mid\mid\mid \sqcup \\
&\xrightarrow{(9)} \sqcup q_0 \sqcup \mid\mid\mid\mid\mid \sqcup \\
&\xrightarrow{(1)} \sqcup q_3 \mid\mid\mid\mid\mid\mid \sqcup \\
&\xrightarrow{(7)} \sqcup \mid q_1 \mid\mid\mid\mid\mid \sqcup \\
&\xrightarrow{(4)} \sqcup \mid q_7 \mid\mid\mid\mid\mid \sqcup
\end{aligned}
$$

Der Zustand q_7 ist der Haltezustand. Daher terminiert die `TM`. Die TURING Maschine gibt sechs $\mid$ –Zeichen auf das Band.

A.5.5 Übung [5.53]

Geben Sie eine TM an, die die als Binärzahl interpretierte Bandinschrift $w \in \{0,1\}^*$ in die Zweierkomplementdarstellung umwandelt.

Lösung:

Die Zweierkomplementdarstellung einer Binärzahl bildet man dadurch, dass alle Bits gekippt werden, danach wird 1 addiert. Beispielsweise

$$1011\,0000 \longrightarrow 0100\,1111 \longrightarrow 0101\,0000,$$

oder

$$0\,1101 \longrightarrow 1\,0010 \longrightarrow 1\,0011.$$

Wir können dies folgendermaßen umsetzen, informell arbeitet die TM wie folgt:

- Wenn das letzte Bit eine 1 ist, schreibe eine 1 und invertiere alle Bits links davon.
- Wenn das niedrigwertigste Bit eine 0 ist, dann schreibe eine 0. Gehe dann nach links, bis die erste 1 kommt. Invertiere alle Bits die links von der 1 auftreten.

Die TURING Maschine, die dies umsetzt, ist:

$$M = (Q, \Sigma, \delta, q^s, q_e)$$

mit den fünf Zuständen

$$Q = \{q_0, q_1, q_2, q_3, q_e\},$$

dem Inputalphabet $\Sigma = \{0,1\}$, dem Startzustand q_0, und dem Acceptzustand q_2. Die Übergänge δ sind:

(1)	$q_0 1$	$\longmapsto$	$q_0 R$	nach rechts bis zum $\sqcup$ hinter dem Input
(2)	$q_0 0$	$\longmapsto$	$q_0 R$	nach rechts bis zum $\sqcup$ hinter dem Input
(3)	$q_0 \sqcup$	$\longmapsto$	$q_1 L$	Ende erreicht, eine Zelle nach links
(4)	$q_1 0$	$\longmapsto$	$q_1 L$	
(5)	$q_1 1$	$\longmapsto$	$q_2 L$	
(6)	$q_2 0$	$\longmapsto$	$q_3 1$	
(7)	$q_2 1$	$\longmapsto$	$q_3 0$	
(8)	$q_3 0$	$\longmapsto$	$q_2 L$	
(9)	$q_3 1$	$\longmapsto$	$q_2 L$	
(10)	$q_2 \sqcup$	$\longmapsto$	$q_e \sqcup$	

Beispiel: Wir bilden das Zweierkomplement des Strings $w = \texttt{1010 1000}$.

$$
\begin{array}{rcl}
q_0\texttt{1010 1000} & \xrightarrow{(1)} & \sqcup\texttt{1}q_0\texttt{010 1000} \\
 & \xrightarrow{(2)} & \sqcup\texttt{10}q_0\texttt{10 1000} \\
\vdots & \vdots & \vdots \\
 & \xrightarrow{(3)} & \sqcup\texttt{1010 100}q_1\texttt{0} \\
 & \xrightarrow{(4)} & \sqcup\texttt{1010 10}q_1\texttt{00} \\
 & \xrightarrow{(4)} & \sqcup\texttt{1010 1}q_1\texttt{000} \\
 & \xrightarrow{(4)} & \sqcup\texttt{1010 }q_1\texttt{1000} \\
 & \xrightarrow{(5)} & \sqcup\texttt{101}q_2\texttt{0 1000} \\
 & \xrightarrow{(6)} & \sqcup\texttt{101}q_3\texttt{1 1000} \\
 & \xrightarrow{(9)} & \sqcup\texttt{10}q_2\texttt{11 1000} \\
 & \xrightarrow{(7)} & \sqcup\texttt{10}q_3\texttt{01 1000} \\
 & \xrightarrow{(8)} & \sqcup\texttt{1}q_2\texttt{001 1000} \\
 & \xrightarrow{(6)} & \sqcup\texttt{1}q_3\texttt{101 1000} \\
 & \xrightarrow{(9)} & \sqcup q_2\texttt{1101 1000} \\
 & \xrightarrow{(7)} & \sqcup q_3\texttt{0101 1000} \\
 & \xrightarrow{(8)} & q_2 \sqcup \texttt{0101 1000} \\
 & \xrightarrow{(10)} & q_e \sqcup \texttt{0101 1000}
\end{array}
$$

Output: `0101 1000`

Beispiel: Wir bilden das Zweierkomplement des Strings $w = \texttt{0 1101}$.

$$
\begin{array}{rcl}
q_0\texttt{0 1101} & \xrightarrow{(1)} & \sqcup\texttt{0}q_0\texttt{1101} \\
& \xrightarrow{(2)} & \sqcup\texttt{01}q_0\texttt{101} \\
\vdots & \vdots & \vdots \\
& \xrightarrow{(3)} & \sqcup\texttt{0 110}q_1\texttt{1} \\
& \xrightarrow{(5)} & \sqcup\texttt{0 11}q_2\texttt{01} \\
& \xrightarrow{(6)} & \sqcup\texttt{0 11}q_3\texttt{11} \\
& \xrightarrow{(9)} & \sqcup\texttt{0 1}q_2\texttt{111} \\
& \xrightarrow{(7)} & \sqcup\texttt{0 1}q_3\texttt{011} \\
& \xrightarrow{(8)} & \sqcup\texttt{0}q_2\texttt{1011} \\
& \xrightarrow{(7)} & \sqcup\texttt{0}q_3\texttt{0011} \\
& \xrightarrow{(8)} & \sqcup q_2\texttt{0 0011} \\
& \xrightarrow{(6)} & \sqcup q_3\texttt{1 0011} \\
& \xrightarrow{(9)} & q_2 \sqcup \texttt{1 0011} \\
& \xrightarrow{(10)} & q_e \sqcup \texttt{1 0011}
\end{array}
$$

Output: `1 0011`

A.5.6 Übung [5.54]

Konstruieren Sie eine TURING Maschine M, die rechts neben der Eingabe $w \in \{\mathtt{0},\mathtt{1}\}^*$ nochmals das Wort w schreibt (Kopiermaschine).

Lösung:

Informell können wir die Verarbeitung, die die TURING Maschine ausführt, folgendermaßen beschreiben.

- Gestartet wird links am Wort w.
- Wird eine `1` gelesen, dann ersetze dies durch das Zeichen e. Geh ganz nach rechts und schreibe ein e in die leere Zelle.
- Wird eine `0` gelesen, dann ersetze dies durch das Zeichen n. Geh ganz nach rechts und schreibe ein n in die leere Zelle.
- Geh wieder nach links bis zum e oder n, geh eine Zelle nach rechts und wiederhole bei jeder gelesenen `0` oder `1` diese Schritte.
- Wenn keine Zeichen `0,1` mehr auf dem Band liegen, geh an den Anfang und ersetze jede e durch `1` und jedes n durch `0`.

Literaturverzeichnis

[1] John D. Barrow
Die Entdeckung des Unmöglichen
Spektrum Akademischer Verlag GmbH
Heidelberg, Berlin, 2001.

[2] George S. Boolos, John P. Burgess and Richard C. Jeffrey
Computability and Logic
Fifth Edition
Cambridge University Press, Cambridge, 2010.

[3] Werner Brecht
Theoretische Informatik
Grundlagen und praktische Anwendungen
Friedrich Vieweg Verlag
Braunschweig 1995.

[4] Gregory J. Chaitin
Grenzen der Berechenbarkeit
Spektrum der Wissenschaft
Februar 2004, S. 86 - 93.

[5] B. Jack Copeland (ed.)
The essential Turing
The ideas that gave birth to the computer age
Clarend Press Oxford, 2004.

[6] B. Jack Copeland
Turing
Poineer of the Computer Age
Oxford University Press, 2012.

[7] Liesbeth De Mol
Turing Machines
The Stanford Encyclopedia of Philosophy
Winter 2019 Edition
Edward N. Zalta (ed.)
https://plato.stanford.edu/archives/win2019/entries/turing-machine/

[8] A.K. Dewdney
Der Turing Omnibus
Ein Reise durch die Informatik in 66 Stationen
Springer Verlag, Berlin, 1995.

[9] Martin Gardner
Penrose Tiles to Trapdoor Ciphers
W. H. Freeman and Company, New York, 1989.

[10] David Harel
Das Affenpuzzle
und weitere bad news aus der Computerwelt
Springer Verlag, Berlin, Heidelberg, 2002.

[11] David Harel und Yishai Feldman
Algorithmik
Die Kunst des Rechnens
Springer Verlag, Berlin, Heidlberg, 2006.

[12] Rolf Herken (Ed.)
The Universal Turing Machine
A Half–Century Survey
Springer–Verlag, Wien, New York, 1995.

[13] David Hilbert
Die Hilbertschen Probleme
Ostwalds Klassiker der exakten Wissenschaften
Band 252
Verlag Harri Deutsch, 1998.

[14] Andrew Hodges
Alan Turing, Enigma
Springer–Verlag, Wien, New York, 1989

[15] Dirk W. Hoffmann
Grenzen der Mathematik
Eine Reise durch die Kerngebiete der mathematischen Logik
Spektrum, Heidelberg, 2011.

[16] Douglas Hofstadter
Gödel, Escher, Bach
ein Endloses Geflochtenes Band
Klett Cotta, Stuttgart, 1985.

[17] John E. Hopcroft
Turingmaschinen
Spektrum der Wissenschaften, Juli 1984
Seite 34.

[18] Herveé Lehning
Cantors Diagonale
Spektrum der Wissenschaft
Spezial 2/05: Unendlich (Plus 1), 48 – 51.

[19] Alan P. Parkes
A Concise Introduction to Languages and Machines
Springer, UTCS, London, 2008.

[20] Roger Penrose
The Emperor's New Mind
Concerning Computers, Minds, and the Laws of Physics
Oxford University Press
New York, 1989.
Deutsche Übersetzung:
Computerdenken
Die Debatte um Künstliche Intelligenz, Bewußtsein und die Gesetze der Physik
Spektrum der Wissenschaft Verlagsgesellschaft, Heidelberg.

[21] Charles Petzold
The Annotated Turing
A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine
Wiley Publishing, Indianapolis, Indiana, 2008.

[22] Constance Reid
Hilbert
Springer Verlag
New York, 1996.

[23] Borut Robic
The Foundations of Computability Theory
Springer, Berlin, Heidelberg, 2015.

[24] U. Schöning
Ideen der Informatik
Grundlegende Modelle und Konzepte
Oldenbourg Verlag
München, Wien, 2002.

Automatentheorie

[25] Alfred V. Aho, Jeffrey D. Ullman
Foundation of Computer Science
C Edition
W. H. Freeman and Company
New York, 1995.

[26] Alexander Axelroth, Christel Baier
Theoretische Informatik
Eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen mit 101 Beispielen
Pearson Studium
München, 2002.

[27] Norbert Blum
Theoretische Informatik
Eine anwendungsorientierte Einführung
Oldenbourg Wissenschaftsverlag
München, 1998.

[28] Daniel I.A. Cohen
Introduction to Computer Theory
Second Edition
John Wiley & Sons, Inc.
New Jersey, 1997.

[29] Martin D. Davis, Ron Sigal, Elaine J. Weyuker
Computability, Complexity, and Languages
Fundamentals of Theoretical Computer Science
Second Edition
Morgan Kaufmann Publishing
San Diego, 1994.

[30] Katrin Erk, Lutz Priese
Theoretische Informatik
Eine umfassende Darstellung
Springer Verlag
Berlin, Heidelberg, New York, 2000.

[31] Ulrich Hedtstück
Einführung in die Theoretische Informatik
Formale Sprachen und Automatentheorie
Oldenbourg Wissenschaftsverlag
München, 2000.

[32] W.M.L. Holcombe
Algebraic automata theory

Cambridge studies in advanced mathematics 1
Cambridge University Press
Cambridge, 2004.

[33] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman
Introduction to Automata Theory, Languages and Computation
Second Edition
Addison Wesley
Reading Massachusetts, 2001. New York, Berlin, Heidelberg, 1997.

[34] John E. Hopcroft, Jeffrey D. Ullman
Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie
Addison Wesley
Bonn, Paris, Reading Massachusetts, 1996.

[35] Zvi Kohavi, Niraj K. Jha
Switching and Finite Automata Theory
Third Edition
Cambridge University Press
Cambridge, 2010.

[36] Thomas Koshy
Discrete Mathematics with Applications
Elsevier Academic Press
Amsterdam, 2004.

[37] Dexter C. Kozen
Automata and Computability
Undergraduate Texts in Computer Science
Springer Verlag

[38] Dexter C. Kozen
Theory of Computation
Springer, London, 2006.

[39] Peter Linz
An Introduction to Formal Languages and Automata
Fourth Edition
Jones and Bartlett Publishers
Boston, Toronto, Lindon, Singapore, 20006.

[40] John C. Martin
Introduction to Languages and the Theory of Computation
Fourth Edition
McGraw–Hill, New York, 2011.

[41] Marvin L. Minsky
Computation Finite and Infinite Machines

Prentice Hall
Englewood Cliffs, N.J., 1967.

[42] Marvin Minsky
Berechnung: Endliche und unendliche Maschinen
Verlag berliner Union GmbH, Stuttgart, 1971.

[43] Matthew Simon
Automata Theory
World Scientific
Singapore, New Jersey, London, 2001.

[44] Michael Sipser
Introduction to the Theory of Computation
PWS Publishing Company, Boston, 1997.

[45] Thomas A. Sudkamp
An Introduction to the Theory of Computer Science Languages and Machines
Third Edition
Pearson Eduction, Boston, 2006.

[46] Gottfried Vossen, Kurt–Ulrich Witt
Grundlagen der Theoretische Informatik mit Anwendungen
Eine Einführung für Studierende der Informatik, Wirtschaftsinformatik und Technischen Informatik
Friedrich Vieweg Verlag
Braunschweig 2000.

[47] Ingo Wegener
Theoretische Informatik
Eine algorithmenorientierte Einführung
3. Auflage
Teubner, Wiesbaden, 2005.

Komplexitätstheorie

[48] Michael R. Garey and David S. Johnson
Computers and Intractability
A Guide to the Theory of NP–Completeness
W. H. Freeman and Company
New York, 1979.

[49] Christos H. Papadimitriou
Computational Complexity
Addison Wesley
Reading Massachusetts, 1995.

[50] Ingo Wegener
Komplexitätstheorie
Grenzen der Effizienz von Algorithmen
Springer Verlag
Berlin, Heidelberg, New York, 2003.

[51] Sanjeev Arora and Boaz Barak
Computational Complexity
A Modern Approach
Cambridge University Press, Cambridge, 2009.

Verschiedenes

[52] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullmann
Compilerbau, Teil 1
2. Auflage
Oldenbourg, München, 1999.

[53] Alfred V. Aho, Brian W. Kerninghan, Peter J. Weinberger
The AWK Programming Language
Addison Wesley Publishing Company
Reading, Massachusetts, 1988.

[54] Alexander Asteroth, Christel Baier
Theoretische Informatik
Eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen mit 101 Beispielen
Pearson, München, 2002.

[55] Richard Courant, Herbert Robbins
Was ist Mathematik?
5. Auflage
Springer Verlag
Berlin, Heidelberg, New York, 2001.

[56] Martin Davis (ed.)
The Undecidable
Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions
Dover Publications, Inc.
New York, 1965.

[57] Martin Davis
Computability and Unsolvability
Dover Publications, Inc.
New York, 1982.

[58] Oliver Deussen, Bernd Lintermann
Computerpflanzen
Spektrum der Wissenschaft
Februar 2001, p. 58 – 65.

[59] Keith Devlin
The Millenium Problems
The Seven Greatest Unsolved Mathematical Puzzeles of Our Time
Basic Books, New York, 2002.

[60] Dale Dougherty & Arnold Robbins
sed & awk
2nd Edition

O'Reilly & Associates, Inc.
Cambridge, Köln, Paris, Sebastopol, 1997.

[61] Richard P. Feynman
Feynman Lectures on Computation
Westview Press
Boulder, 2000.

[62] Jeffrey E. F. Friedl
Reguläre Ausdrücke
O'Reilly & Associates, Inc.
Cambridge, Köln, Paris, Sebastopol, 2001.

[63] Rowan Garnier, John Taylor
Discrete Mathematics for New Technology
Second Edition
Institute of Physics Publishing
Bristol and Philadelphia, 2002.

[64] Ronald L. Graham, Donald E. Knuth, Oren Patashnik
Concrete Mathematics
A Foundation for Computer Science
Second Edition
Addison Wesley
Reading, Massachusetts, 1994.

[65] Jozef Gruska
Foundations of Computing
International Thompson Computer Press, 1997.

[66] Helmut Herold
awk & sed
Die Profitools zur Dateibearbeitung und –editierung
Addison Wesley Verlag
München, Boston, 2003.

[67] Helmut Herold
lex & yacc
Die Profitools zur lexikalischen und syntaktischen Textanalyse
Addison Wesley Verlag
München, Boston, 2003.

[68] J. Roger Hindley and Jonathan P. Seldin
Lambda–Calculus and Combinators
An Introduction
Cambridge University Press, 2008.

[69] Thomas Hinze, Monika Sturm
Rechnen mit DNA

Ein Einführung in Theorie und Praxis
Oldenbourg Verlag, München, Wien, 2004.

[70] Hartmut Jürgens, Heinz–Otto Peitgen, Dietmar Saupe
Fraktale — eine neue Sprache für komplexe Strukturen
Spektrum der Wissenschaft
September 1989, p. 52 – 64.

[71] Lukas König, Friederike Pfeiffer–Bohnen, Hartmut Schmeck
100 Übungsaufgaben zu Grundlagen der Informatik
Band 1: Theoretische Informatik
Oldenbourg Verlag, München, 2014.

[72] John R. Levine, Tony Mason & Doug Brown
lex & yacc
O'Reilly & Associates, Inc.
Cambridge, Köln, Paris, Sebastopol, 1995.

[73] Harry R. Lewis, Christos H. Papadimitriou
The Efficiency of Algorithms
Scientific American
Vol. **238**, 1 (1978), pp. 96 – 109.

[74] Benoit B. Mandelbrot
Die fraktale Geometrie der Natur
Birkhäuser Verlag
Basel, Boston, Berlin, 1991.

[75] Marvin L. Minsky, Seymour Papert
Perceptrons
Expanded Edition
The MIT Press, Cambridge, Massachusetts, 1990.

[76] Heinz–Otto Peitgen, Hartmut Jürgens, Dietmar Saupe
Chaos and Fractals
New Frontiers of Science
Springer Verlag
New York, Berlin, Heidelberg, 1992.

[77] György E. Revesz
Introduction to Formal Languages
Dover Publications, Inc., New York, 1983.

[78] Arnold Robbins
Effective awk Programming
3rd Edition
O'Reilly & Associates, Inc.
Cambridge, Köln, Paris, Sebastopol, 2001.

[79] Grzegorz Rozenberg and Arto Salomaa
Cornerstones of Undecidability
Pearson, 2001.

[80] Uwe Schöning
Theoretische Informatik – kurz gefasst
5. Auflage
Spektrum, Heidelberg, 2009.

[81] Raymond M. Smullyan and Melvin Fitting
Set Theory and the Continuum Problem
Dover Publications, Mineola, 2010.

[82] Larry J. Stockmeyer, Ashok K. Chandra
Intrinsically Difficult Problems
Scientific American
Vol. **240**, 5 (1979) pp. 124 – 133.

[83] Benjamin H. Yandell
The Honors Class
Hilbert's Problems and Their Solvers
A. K. Peters
Natick, Massachusetts, 2001.

[84] Marcus du Sautoy
Die Musik der Primzahlen
C.H. Beck Verlag
München, 2004.

[85] Jeffrey D. Ullman
Fundamental Concepts of Programming Systems
Addison–Wesley, Reading, Massachusetts, 1976.

Papers

[86] Theodore Baker, John Gill, and Robert Solovay
Relativizations of the $\mathcal{P} =? \mathcal{NP}$ Question
SIAM Journal of Computation **4** (1975), 431 – 442.

[87] Y. Bar–Hillel, M. Perles, and E. Shamir
On formal properties of simple phrase-structure grammars
Z. Phonetik. Sprachwiss. Kommunikationsforsch. **14** (1961), pp. 143 – 172.

[88] Y. Bar–Hillel, M. Perles, and E. Shamir
On Formal Properties of Simple Phrase Structure Grammars
in Y. Bar-Hillel (ed.) *Language and Information*
Addison-Wesley,
Reading, MA (1964), pp. 116 – 150.

[89] R. Berger
The Undecidability of the Domino Problem
Memoirs Amer. Math. Soc. **66** (1966).

[90] Noam Chomsky
Three models for the description of a language
IRE transactions on Information Theory
2: 3 (1956), 113–124.

[91] Alonzo Church
A set of postulates for the foundations of logic
Ann. Math. **33–34** (1933), 346 – 366, 839 – 864.

[92] Alonzo Church
An unsolvable problem of elementary number theory
Amer. J Math. **58** (1936), 345 – 363.

[93] Barry Cipra
Wer wird Millionär?
Omega, 4/2003, pp. 40.

[94] Stephen A. Cook
The Complexity of Theorem–Proving Procedures
In: *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*
1971, pp. 151 - 158.

[95] Stephen A. Cook
An overview of computational complexity
Communications of the ACM, **26** (1983), Vol. 6, pp. 401 – 408.

[96] Huskell B. Curry
An analysis of logical substitutions
Amer. J. Math. 51 (1929), pp.363 – 384.

[97] Martin Davies and Reuben Hersh
Hilbert's 10th Problem
Scientific American, 229, No. 5, p. 84 (1973).

[98] Edsger W. Dijkstra
Notes on Structured Programming
In: Software Engineering
Academic Press
London, 1972, S. 1 - 82.

[99] Lance Fortnow
The Status of the P versus NP Problem
Communications of the ACM, Vol. 52, No. 9, Sept. 2009, pp. 78 – 88.

[100] Lance Fortnow and Steve Homer
A short history of computational complexity
Bulletin of the European Association for Thepretical Computer Science 80, (June 2003).

[101] Kurt Gödel
Über formal unentscheidbare Sätze der Principia Mathematica und verwandte Systeme I
Monatshefte für Mathematik und Physik
38 (1931), 173 - 198.

[102] Kurt Gödel
On undecidable propsitions of formal mathematical systems
in: Martin Davis, [56], pp. 41.

[103] Martin Grötschel, Manfred Padberg
Die optimierte Odysee
Spektrum der Wissenschaft
Heft April 1999, Seite 76.

[104] Richard Kaye
Minesweeper is NP–complete
The Mathematical Intelligencer
Vol. **22**, No. 2, 2000. pp 9 - 15.

[105] Stephen C. Kleene
A theory of positive integers in formal logic
Amer. J. Math. **57**, (1935), pp. 153–173, 219–244.

[106] Dexter Kozen
On Kleene Algebras and closed Semirings
In Rovan, editor, *Prc. Math. Found. Comput. Sci. 1990* volume 452 of *Lect. Notes in Comput. Sci.*, pages 26-47. Springer Verlag, 1990.

[107] Christoph Lampert
Ukrainische Überraschung
Theoretische Informatik: Gilt doch $\mathcal{P} = \mathcal{NP}$?
c't – Magazin für Computertechnik
Heft 22/2000, Seite 62.

[108] Hing Leung
Two–Way deterministic Finite Automaton
http://www.math.nmsu.edu/hist_projects/2DFA.pdf.

[109] Yuri Matijasevic
Enumerable sets are Diophantine
Soviet Mathematics Doklady
11. (1970), 354–357.

[110] George H. Mealy
A method for synthezising sequential circuits
Bell System Technical Journal
34, 1045 (1955).

[111] Marvin Minsky
Recursive Unsolvability of Post's problem of 'Tag' and other Topics in Theory of Turing Machines
Annals of Mathematics, Vol. 74, No. 3, Nov. 1961.

[112] Edward F. Moore
Gedanken–experiments on sequential machines
In *Automata Studies*, ed. by C.E. Shannon and J. McCarthy, Princeton University Press, Pronceton, N.J. pp. 129–153, (1956).

[113] John Myhill
Finite Automata and the representations of events
WADD TR–57–624, pp. 112 – 137 (1957).

[114] Anil Nerode
Linear Automaton Transformations
Proc. AMS **9**, 541 – 544 (1958).

[115] Emil Post
Finite combinatory processes–formulation I
J. Symbolic Logic, **1**, (1936), pp. 103–105.

[116] Emil Post
Formal reductions of the general combinatorial decision problem
Am. J. Math. **43**, (1943), pp. 197–215.

[117] Emil Post
A variant of a recursively unsolvable problem
Bulletin of the American Mathematical Society **52**, (1946), pp. 264–268.

[118] Micheal O. Rabin, Dana Scott
Finite Automata and Their Decision Problems
IBM Jour. Res. Develop. 3 (1959), pp. 115 – 125.

[119] Steffen Reith, Heribert Vollmer
Wer wird Millionär?
Komplexitätstheorie: Konzepte und Herausforderungen
c't - Magazin für Computertechnik
Heft 7/2001, Seite 240 – 251.

[120] Moses Schönfinkel
Über die Bausteine der mathematischen Logik
Math. Annalen, 92 (1924), pp. 305 – 316.

[121] J.C. Shepherdson
The reduction of two–way automata to one–way automata
IBM Jour. Res. Develop. 3 (1959), pp. 198 – 200.

[122] J. Shepherdson, H.E. Sturgis
Computability of Recursive Functions
J. ACM **10**, 217 – 255, 1963.

[123] Michael Sipser
The History and status of the P versus NP Question

[124] Ian Stewart
Wer wird Millionär
Spektrum der Wissenschaft
Heft Mai 2002, Seite 114 – 115.

[125] Alan M. Turing
On computable numbers with an application to the Entscheidungsproblem
Proc. London Mathematical Society
42 (1937), 230 - 265.
Korrektur in **43**,544 - 546.

[126] Hua Wang
Proving Theorems by Pattern Recognition
Bell Syst. Tech. J. **40** (1981), pp. 1 – 42.

Index