Mathematik und Logik und Formale Grundlagen 2006/7

Franz Binder

30. Mai 2007

Inhaltsverzeichnis

Kapitel 9

Automaten, Formale Sprachen, Berechenbarkeit

9.1 Wörter

9.1.1 Notation. Für jede Menge Σ, bezeichne Σ^* die Menge aller Listen von Objekten vom Typ Σ. Wenn Σ endlich ist, nennt man deren Elemente gelegentlich *Zeichen* oder *Symbole*, und Σ heißt dann ein *Alphabet*, und die Elemente von Σ^* heißen dann *Zeichenketten* (*Strings*) oder *Wörter* über dem Alphabet Σ. Die Länge eines Wortes w wird mit $|w|$ bezeichnet. Die leere Liste heißt dann das *leere Wort* und wird meist mit ϵ bezeichnet. Die Verkettung von Wörtern wird meist ohne Symbol, einfach durch Nebeneinanderstellen, bezeichnet ($u \diamond v = uv$). Wir verwenden auch Potenznotation ($u^0 = \epsilon$, $u^{n+1} = u^n u$). Ferner steht jedes Symbol gleichzeitig als Abkürzung für das entsprechende Wort der Länge 1 ($\alpha \triangleleft u = \alpha u$, $\Sigma \subseteq \Sigma^*$). Zur besseren Lesbarkeit und der Einfachheit halber bezeichnen wir hier Symbole zumeist mit kleinen griechischen Buchstaben ($\alpha, \beta, \gamma, \ldots$) und Wörter mit kleinen lateinischen Buchstaben ($u, v, w, \ldots$).

9.1.2 Bemerkung. So wie jede Listenverkettung ist auch die Wortverkettung eine assoziative Operation, und das leere Wort ist neutrales Element:

$$(uv)w = u(vw); \tag{9.1}$$

$$u\epsilon = u = \epsilon u. \tag{9.2}$$

Die Wörter über Σ bilden daher ein Monoid, das *freie Monoid* über Σ.

9.1.3 Satz. *Für jedes Alphabet Σ ist die Wortlänge ein Homomorphismus in das additive Monoid der natürlichen Zahlen $(\mathbb{N}, +)$, d.h.*

$$|\epsilon| = 0; \tag{9.3}$$

$$|uv| = |u| + |v|\,. \tag{9.4}$$

9.1.4 Bemerkung. Funktionen auf Wörtern werden so wie für Listen üblich typischerweise rekursiv definiert.

9.1.5 Beispiel. Die *Spiegelung* eines Wortes wird definiert durch: $s\colon \Sigma^* \to \Sigma^*$,

$$s(\epsilon) = \epsilon, \tag{9.5}$$

$$s(\alpha w) = s(w)\alpha. \tag{9.6}$$

9.1.6 Beispiel. Ein *Palindrom* ist ein Wort w, für welches $w = s(w)$ gilt.

9.1.7 Theorem. *Sei $\mathcal{M}$ ein beliebiges Monoid. Dann läßt sich jede Abbildung $f\colon \Sigma \to \mathcal{M}$ zu genau einem Homomorphismus $f^*\colon \Sigma^* \to \mathcal{M}$ fortsetzen.*

Beweis. Jeder derartige Homomorphismus muß die folgenden Gleichungen erfüllen.

$$f^*(\epsilon) = \epsilon, \tag{9.7}$$
$$f^*(\alpha w) = f(\alpha) f^*(w). \tag{9.8}$$

Tatsächlich wird dadurch eindeutig eine Funktion definiert, und man kann mit Induktion zeigen, daß diese tatsächlich ein Homomorphismus ist. □

9.2 Formale Sprachen

9.2.1 Definition. Sei Σ ein Alphabet. Jede Teilmenge von Σ^* nennt man eine *formale Sprache* über Σ.

9.2.2 Bemerkung. Da Sprachen Teilmengen sind, stehen auch für Sprachen die üblichen Mengenoperationen zur Verfügung, insbesondere Vereinigung, Durchschnitt, Mengendifferenz. Darüberhinaus können weitere Operationen für Sprachen definiert werden, die sich aus der Übertragung von Operationen für Wörter ergeben.

9.2.3 Definition. Seien L, L_1, L_2 Sprachen über dem Alphabet Σ. Dann definiert man

$$L_1 L_2 = \{\, uv \mid u \in L_1,\ v \in L_2 \,\} \qquad (\textit{Verkettung}) \tag{9.9}$$
$$L^0 = \{\, \epsilon \,\} \qquad (\text{einelementig}) \tag{9.10}$$
$$L^{n+1} = LL^n \qquad (\text{für } n\colon \mathbb{N}) \tag{9.11}$$
$$L^* = \bigcup_{k\colon \mathbb{N}} L^k \qquad (\textit{Iteration}) \tag{9.12}$$

9.2.4 Bemerkung. Man beachte: $L^2 = LL = \{\, uv \mid u \in L,\ v \in L \,\}$, was im allgemeinen wesentlich mehr Elemente enthält als $\{\, uu \mid u \in L \,\}$.

9.2.5 Bemerkung. Mit dieser Verkettung von Sprachen ist somit auch auf der Menge $\mathscr{P}(\Sigma^*)$ aller Sprachen über einem Alphabet eine Monoidstruktur definiert.

9.2.6 Beispiel. Sei Δ ein weiteres Alphabet. Dann nennt man eine Abbildung der Form $f\colon \Sigma \to \mathscr{P}(\Delta^*)$ eine *Substitution*; sie wird natürlich ebenfalls auf Σ^* fortgesetzt zu $f^*\colon \Sigma^* \to \Delta^*$ mittels

$$f^*(\epsilon) = \epsilon, \tag{9.13}$$
$$f^*(\alpha w) = f(\alpha) f^*(w). \tag{9.14}$$

9.2.7 Beispiel. Ist $h\colon \Sigma^* \to \Delta^*$ ein Homomorphismus und sind $L_1\colon \mathscr{P}(\Sigma^*)$, $L_2\colon \mathscr{P}(\Delta^*)$ Sprachen, dann heißt die Sprache $h(L) = \{\, h(w) \mid w \in L \,\}$ ein *homomorphes Bild* von L, und die Sprache $h^{-1}(L) = \{\, w\colon \Sigma^* \mid h(w) \in L \,\}$ ist ein *inverses homomorphes Bild*.

Für jedes $\alpha\colon \Sigma$ ist $\{\,\alpha\,\}$ eine Sprache; sie besteht nur aus einem Wort. Noch trivialer sind die Sprachen $\{\,\epsilon\,\}$ und $\varnothing$.

Jede endliche Sprache läßt mit Verkettung und Vereinigung aus diesen trivialen Sprachen bilden. Außerdem ist jede Verkettung oder Vereinigung endlicher Sprachen wieder endlich. Die endlichen Sprachen lassen sich somit auch als jene Klasse von Sprachen charakterisieren, welche gerade groß genug ist, daß sie

- die oben erwähnten trivialen Sprachen enthält,
- gegenüber Verkettung abgeschlossen ist, und
- gegenüber Vereinigung abgeschlossen ist.

Zusätzlich ist die Klasse aller endlichen Sprachen gegenüber Durchschnitt und Differenz abgeschlossen, nicht aber gegenüber Komplement und Iteration.

9.3 Reguläre Ausdrücke

9.3.1 Definition. Die kleinste Klasse von formalen Sprachen, welche alle endlichen Sprachen umfaßt und abgeschlossen ist gegenüber Verkettung, Vereinigung und Iteration, heißt die Klasse der *regulären Sprachen*.

Reguläre Sprachen werden zumeist durch *reguläre Ausdrücke* (regular expression, regex) beschrieben:

9.3.2 Definition. Sei Σ ein Alphabet. Dann ist jedes $\alpha \in \Sigma$ ein regulärer Ausdruck, und wenn r, s reguläre Ausdrücke sind, dann auch (rs), $(r|s)$ und r^*, entsprechend den definierenden Konstruktionen für reguläre Sprachen.

Natürlich kann man bei regulären Ausdrücken auf Klammern verzichten, wenn der Aufbau auch ohne diese klar ist.

9.3.3 Definition. Jedem regulären Ausdruck r wird gemäß der folgenden induktiven Definition eine reguläre Sprache $L(r)$ zugeordnet:

$$L(\alpha) = \{\,\alpha\,\} \tag{9.15}$$
$$L(rs) = L(r)\,L(s) \tag{9.16}$$
$$L(r|s) = L(r) \cup L(s) \tag{9.17}$$
$$L(r^*) = L(r)^* \tag{9.18}$$

9.3.4 Bemerkung. Darüberhinaus verwenden die gängigen Programme zur Verarbeitung von regulären Ausdrücken aus praktischen Gründen diverse Abkürzunge z.B.

$$\begin{aligned} r? &= (r|\epsilon) \\ r^+ &= rr^* \\ [xyz] &= x|y|z \\ . &= \alpha|\beta|\gamma|\ldots|\zeta, \text{ falls } \Sigma = \{\,\alpha,\beta,\gamma,\ldots,\zeta\,\} \\ [c{-}u] &= c|d|\ldots|t|u \\ [\hat{}xyz] &= l_1|l_2|l_3|\ldots|l_n, \text{ falls } \Sigma \setminus \{x,y,z\} = \{l_1,\ldots,l_n\} \end{aligned}$$

und viele mehr. Details dazu findet man z.B. in
http://en.wikipedia.org/wiki/Regular_expression.
Einen guten Vergleich der Syntaxregeln für reguläre Ausdrücke in verschiedenen Programmiersprachen bietet:
http://www.greenend.org.uk/rjk/2002/06/regexp.html

9.4 Endliche Automaten

9.4.1 Notation. Sei Σ eine beliebige Menge (von Symbolen) und Q eine Menge (von Zuständen). Eine Funktion $\delta\colon \Sigma \to (Q \to Q \to \Omega)$, welche jedem Symbol eine binäre Relation in der Zustandsmenge Q, zuordnet nennt man eine *Zustandsüberführungsrelation*. Die Anwendung von δ auf ein Symbol $\alpha\colon \Sigma$ sei mit δ_α bezeichnet. Jedes δ_α ist somit eine Relation in Q, also ein Digraph (mit Knotenmenge Q, ohne Mehrfachkanten, möglicherweise mit Schlingen). Da die Digraphen für $\delta_\alpha, \delta_\beta, \delta_\gamma, \dots$ (für $\alpha, \beta, \gamma, \dots : \Sigma$) alle dieselbe Knotenmenge haben, ist es sinnvoll, all diese Digraphen in einem einzigen Digraphen zu überlagern. Die Kanten aus den einzelnen Teilen werden dabei zur Unterscheidung mit den zugehörigen Symbolen $(\alpha, \beta, \gamma, \dots)$ gekennzeichnet. Dadurch ergibt sich der *Zustandsgraph* von δ, ein bewerteter Digraph mit Mehrfachkanten (welche sich allerdings stets durch die Kennzeichnung unterscheiden).

Die Existenz einer mit α gekennzeichneten Kante von i nach j ($i \xrightarrow{\alpha} j$) wird dann gelesen als „Die Eingabe des Symbols α erlaubt einen Übergang vom Zustand i in den Zustand j“. Man sagt auch, j ist ein *Folgezustand* von i.

9.4.2 Notation. Bekanntlich kann jede binäre Relation in Q als eine Funktion vom Typ $Q \to \mathscr{P}(Q)$ aufgefaßt werden. Mit $\delta_\alpha(i)$ bezeichen wir daher die Menge aller Folgezustände bei der Eingabe α, also $\delta_\alpha(i) := \{\, j\colon Q \mid i \xrightarrow{\alpha} j \,\}$. Wir verallgemeinern dies für $J \subseteq Q$ noch zu $\delta_\alpha(J) := \bigcup_{j\in J} \delta_\alpha(j)$, der Menge aller Folgezustände bei der Eingabe α von Zuständen in J.

9.4.3 Bemerkung. Gemäß Satz **??** bildet, für jede Menge Q, die Menge der binären Relationen $Q \to Q \to \Omega$ mit dem Relationenprodukt ein Monoid. Somit kann gemäß 9.1.7 jede Zustandsüberführungsrelation $\delta\colon \Sigma \to (Q \to Q \to \Omega)$ auf natürliche und eindeutige Weise zu einem Homomorphismus $\delta^*\colon \Sigma^* \to (Q \to Q \to \Omega)$ fortgesetzt werden; konkret:

$$\delta^*_\epsilon = \Delta_Q \tag{9.19}$$
$$\delta^*_{\alpha w} = \delta_\alpha ; \delta^*_w \tag{9.20}$$

oder noch konkreter:

$$\delta^*_\epsilon(i) = \{\, i \,\} \tag{9.21}$$
$$\delta^*_{\alpha w}(i) = \delta^*_w(\delta_\alpha(i)) \tag{9.22}$$

für jedes $i\colon Q$.

Die Beziehung $i \xrightarrow{u} j$ (d.h. $j \in \delta^*_u(i)$) entspricht dann im Zustandsgraphen der Existenz eines Kantenzuges, dessen Kanten mit den Zeichen der Zeichenkette u gekennzeichnet sind, und bedeutet: „Die Eingabe der Zeichenkette u erlaubt einen Übergang von Zustand i in den Zustand j“. Man sagt dann auch, j ist von i aus *erreichbar*.

9.4.4 Bemerkung. Sind i und f Zustände eines Automaten, dann bildet die Menge aller Zeichenketten, welche einen Übergang vom Zustand i in den Zustand j erlauben, eine Sprache. Der Zustand i heißt in diesem Zusammenhang *Initialzustand*, während f als *Finalzustand* bezeichnet wird. Diese Idee läßt sich auch auf Zustandsmengen verallgemeinern.

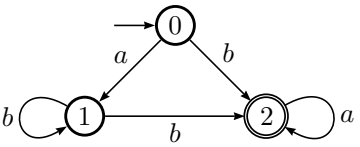
9.4.5 Definition. Ein *nicht-deterministischer, endlicher Automat* besteht aus

- einer endlichen Menge Σ, dem *Eingabealphabet*;
- einer endlichen Menge Q von *Zuständen* (*states*);
- einer entscheidbaren *Zustandsüberführungsrelation* $\delta\colon \Sigma \to (Q \to Q \to \mathbb{B})$;
- einer Teilmenge $I \subseteq Q$ von *Initialzuständen*;
- einer Teilmenge $F \subseteq Q$ von *Finalzuständen*.

Diese Konstruktion wird dann auch mit dem Quintupel $(\Sigma, Q, \delta, I, F)$ bezeichnet.

9.4.6 Notation. Um einen Automaten vollständig zu beschreiben, müssen im Zustandsgraphen noch die Intial- und Finalzustände speziell gekennzeichnet werden. Eine gängige Konvention dazu ist, Initialzustände durch einen Pfeil und Finalzustände durch doppeltes Einkreisen zu kennzeichnen.

9.4.7 Beispiel.

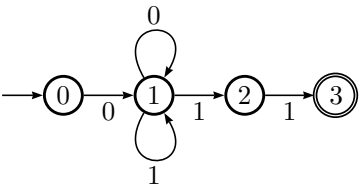


9.4.8 Definition. Die *Sprache* $L(\mathscr{A})$ eines Automaten $\mathscr{A} = (\Sigma, Q, \delta, I, F)$, besteht aus allen Wörtern, die einen Übergang von einem Initialzustand in einen Finalzustand erlauben, d.h.

$$L(\mathscr{A}) := \{\, u\colon \Sigma^* \mid \delta_u^*(I) \cap J \neq \varnothing \,\}.$$

Man sagt auch, der Automat *erkennt* oder *akzeptiert* die Sprache $L(\mathscr{A})$, bzw. jedes Wort in dieser Sprache.

9.4.9 Beispiel. Der endliche Automat



beschreibt dieselbe Sprache wie der reguläre Ausdruck $0(0|1)^*11$.

9.4.10 Bemerkung. Man beachte, daß in dieser Definition die Erreichbarkeit *eines* Finalzustandes ausreicht. Ein akzeptiertes Wort darf somit zusätzlich auch einen Übergang in einen Nicht-Finalzustand erlauben; d.h. es ist z.B. nicht gefordert, daß $\delta_u^*(I) \subseteq J$ gelten müsse.

Da $\delta_\alpha(i)$ durchaus mehrere Elemente enthalten kann, heißen derartige Automaten genauer *nicht-deterministisch*.

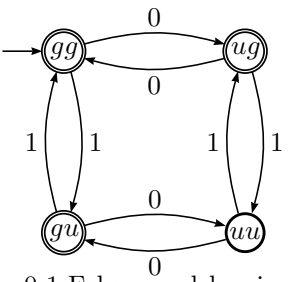
9.5 Deterministische Automaten

9.5.1 Definition. Ein *deterministischer, endlicher Automat* besteht aus

- einer endlichen Menge Q von *Zuständen* (*states*);
- einer endlichen Menge Σ, dem *Eingabealphabet*;
- einer *Zustandsüberführungsfunktion* $\delta\colon \Sigma \to (Q \to Q)$;
- einem Initialzustand $q_0\colon Q$;
- einer Teilmenge $F \subseteq Q$ von *Finalzuständen*.

Diese Konstruktion wird dann auch mit dem Quintupel $(\Sigma, Q, \delta, q_0, F)$ bezeichnet.

9.5.2 Beispiel. Der deterministische Automat



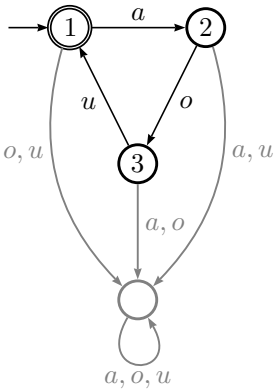
akzeptiert die Sprache aller 0-1-Folgen, welche eine gerade Anzahl von Nullern oder Einsern aufweisen.

9.5.3 Bemerkung. Jeder deterministische Automat kann als Spezialfall eines nicht-deterministischen Automaten aufgefaßt werden, zumal jede Funktion als Relation betrachtet werden kann und jedem Zustand eine einelementige Zustandsmenge entspricht. Dem Relationenprodukt in Gleichung 9.19 entspricht dann die Hintereinanderausführung von Funktionen. Durch $\delta_\alpha(i)$, und auch $\delta_\alpha^*(i)$, werden einzelne Zustände bezeichnet, nicht Mengen von Zuständen.

Zwischenformen: Ein *partieller* (deterministischer) Automat hat zu jedem Input höchstens einen Folgezustand. Und ein *vollständiger* Automat hat zu jedem Input mindestens einen Folgezustand.

Ein partieller Automat kann leicht vervollständigt werden, indem man einen neuen Zustand ω (weder initial noch final), der immer dann als Folgezustand verwendet wird, wenn sonst keiner zur Verfügung steht. Insbesondere ist jeder Folgezustand von ω wieder ω selbst. Man kommt als aus diesem Zustand nie wieder raus, insbesondere erreicht man nie einen Finalzustand. Dieser vervollständigte Automat funktioniert somit praktisch exakt so wie die unvollständige Variante. Insbersondere akzeptieren beide dieselbe Sprache.

9.5.4 Beispiel. Ein patieller deterministischer Automat über dem Alphabet $\{a, o, u\}$



Die Vervollständigung wurde grau dargestellt.

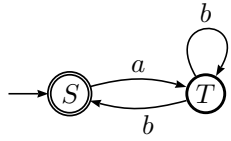
Weiters betrachtet man gelegentlich Automaten mit *spontanen Übergängen* (das sind solche, die keine Eingabe erfordern). Ein derartiger Automat kann stets auf einen funktionsgleichen ohne spontane Übergange zurückgeführt werden.

9.5.5 Satz. *Zu jedem nicht-deterministischen Automaten gibt es einen deterministischen Automaten, der dieselbe Sprache erkennt.*

Beweis. Im Prinzip ersetzt man die Zustandsmenge Q des nicht-deterministischen Automaten durch deren Potenzmenge $\mathscr{P}(Q)$, um die Zustandüberführung künstlich deterministisch zu „machen“. Allerdings sind dann oft die meisten Zustände gar nicht erreichbar, sodaß man auf diese leicht verzichten kann. Die erreichbaren Zustände kann man etwa folgendermaßen finden: Als Initialzustand verwendet man die Menge I der Initialzustände. Dann betrachtet man für jede Eingabe $\alpha \in \Sigma$ die Menge aller Zustände $\delta_\alpha(I)$, in welche der Automat damit gelangen kann. Für jede dieser Mengen X bestimmt man wieder für jede Eingabe α die Menge der dann möglichen Folgezustände $\delta_\alpha(X)$, bis keine neuen Teilmengen mehr gefunden werden. Eine Teilmenge ist dann genau dann ein Finalzustand wenn sie mindestens einen Finalzustand des ursprünglichen Automaten enthält. □

Damit kann man in Zusammenhang mit Sprachen stets einen deterministischen Automaten voraussetzen, und trotzdem auch nicht-deterministische (auch mit spontanen Übergängen) verwenden, wenn das einfacher ist.

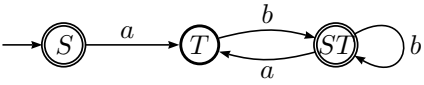
9.5.6 Beispiel. Zum nicht-deterministischen Automaten



bestimmen wir eine deterministische Version:

State	a	b
$\{S\}$	$\{T\}$	
$\{T\}$		$\{S, T\}$
$\{S, T\}$	$\{T\}$	$\{S, T\}$

Damit ergibt sich der partielle deterministische Automat



9.6 Reguläre Sprachen und Automaten

Automaten lassen sich auch verbinden:

9.6.1 Definition. Seien $\mathscr{A}_1, \mathscr{A}_2$ zwei Automaten über demselben Eingabealphabet.

Der Zustandsgraph einer *Parallelschaltung* $\mathscr{A}_1 + \mathscr{A}_2$ besteht aus der Vereinigung der Zustandsgraphen der jeweiligen Automaten. Auch für die Intial- und Finalzustände einer Parallelschaltung von Automaten verwendet man einfach die entsprechenden Vereinigungsmengen.

Bei der *Serienschaltung* $\mathscr{A}_1\mathscr{A}_2$ geht man ähnlich vor, fügt aber spontane Verbindungen von jedem Finalzustand des ersten Akzeptors zu jedem Initialzustand des zweiten hinzu. Die Initialzustände des ersten Automaten werden zu Initialzuständen der Serienschaltung, und deren Finalzustände erhalten wir aus denen des zweiten Automaten.

Für die *Rückkoppelung* $\mathscr{A}^*$ braucht man nur einen Automaten, der durch spontane Übergänge von jedem Finalzustand zu jedem Anfangszustand ergänzt wird; ferner wird die Menge der Finalzustände um den Anfangszustand ergänzt.

9.6.2 Satz. *Seinen $\mathscr{A}_1, \mathscr{A}_2$ endliche Automaten. Dann gilt*

$$\begin{aligned} L(\mathscr{A}_1\mathscr{A}_2) &= L(\mathscr{A}_1)L(\mathscr{A}_2) \\ L(\mathscr{A}_1 + \mathscr{A}_2) &= L(\mathscr{A}_1) \cup L(\mathscr{A}_2) \\ L(\mathscr{A}_1^*) &= L(\mathscr{A}_1)^*. \end{aligned}$$

9.6.3 Satz. *Eine Sprache ist genau dann regulär wenn sie durch einen endlichen Automaten erkannt wird.*

Beweis. Klarerweise kann man zu jeder endlichen Sprache einen erkennenden Automaten bilden. Und Satz 9.6.2 besagt gerade, wie die entsprechenden Automaten für eine Verkettung, Vereinigung oder Iteration einer Sprache zu konstruieren sind.

Sei nun umgekehrt ein Automat $\mathscr{A}$ gegeben. Wir zeigen, daß $L(\mathscr{A})$ regulär ist. Dazu definieren wir zuerst die folgenden Sprachen:
$R^Y_{i,j} := \{w \in \Sigma^* \mid \delta^*_w(i) = j \wedge \delta^*_v(i) \in Y$ für jedes echte Präfix v von w mit $0 < |v| < |$
$R^Y_{i,j}$ besteht damit aus all jenen Wörtern, die vom Zustand i in den Zustand j führen, und dabei *zwischendurch* nur in Zustände aus der Menge Y führen (i, j müssen aber keine Elemente von Y sein). Dann gilt

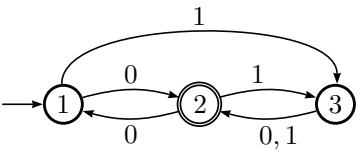
$$L(\mathscr{A}) = \bigcup_{j \in F} R^Q_{0,j}.$$

Wir müssen also noch zeigen, daß jedes $R^Q_{0,j}$ regulär ist. Zu diesem Zweck fahren wir rekusiv folgendermaßen fort

$$R^{Y \cup \{\,k\,\}}_{i,j} = R^Y_{i,j} \cup R^Y_{i,k}(R^Y_{k,k})^* R^Y_{k,j},$$

sodaß wir alles auf die $R^{\varnothing}_{i,j}$ zurückführen, die aber alle endlich sind. □

9.6.4 Beispiel. Gegeben sei der Automat:



Wir bestimmen dazu einen regulären Ausdruck, der dieselbe Sprache bescheibt. Gesucht ist $r_{1,2}$.

$$
\begin{aligned}
r^{\{1,2,3\}}_{1,2} &= r^{\{1,2\}}_{1,2} \mid r^{\{1,2\}}_{1,3}(r^{\{1,2\}}_{3,3})^* r^{\{1,2\}}_{3,2} \\
r^{\{1,2\}}_{1,2} &= r^{\{1\}}_{1,2} \mid r^{\{1\}}_{1,2}(r^{\{1\}}_{2,2})^* r^{\{1\}}_{2,2} \\
&= r^{\{1\}}_{1,2}(r^{\{1\}}_{2,2})^* \\
&= 0(\epsilon \mid 00)^* = 0(00)^* \\
r^{\{1,2\}}_{1,3} &= r^{\{1\}}_{1,3} \mid r^{\{1\}}_{1,2}(r^{\{1\}}_{2,2})^* r^{\{1\}}_{2,3} \\
&= 1 \mid 0(00)^*(1 \mid 01) = 0^*1 \\
r^{\{1,2\}}_{3,3} &= r^{\{1\}}_{3,3} \mid r^{\{1\}}_{3,2}(r^{\{1\}}_{2,2})^* r^{\{1\}}_{2,3} \\
&= \epsilon \mid (0 \mid 1)(00)^*(1 \mid 01) = \epsilon \mid (0 \mid 1)0^*1 \\
r^{\{1,2\}}_{3,2} &= r^{\{1\}}_{3,2}(r^{\{1\}}_{2,2})^* \\
&= (0 \mid 1)(00)^* \\
r^{\{1,2,3\}}_{1,2} &= 0(00)^* \mid 0^*1((0 \mid 1)0^*1)^*(0 \mid 1)(00)^*
\end{aligned}
$$

9.6.5 Satz. *Die Klasse der regulären Sprachen ist gegenüber allen boolschen Mengenoperationen (also auch Durchschnitt und Komplement) sowie gegenüber Spiegelung abgeschlossen.*

Beweis. Es ist nicht offensichtlich, wie ein regulärer Ausdruck für das Komplement einer regulärer Sprache ausschauen soll. Dafür kann man zu jedem Automaten $\mathscr{A}$ ganz einfach jenen konstruieren, der alle Wörter erkennt, die $\mathscr{A}$ nicht erkennt: man verwendet einfach $Q \setminus F$ als Finalzustände. Für den Durchschnitt kann man dann etwa ein De Morgan Gesetz verwenden:

$$L_1 \cap L_2 = \complement(\complement L_1 \cup \complement L_2).$$

Für die Spiegelung dagegen muß man lediglich den regulären Ausdruck spiegeln. □

9.6.6 Bemerkung. Ferner ist die Klasse der regulären Sprachen gegenüber homomorphen Bildern und Substitutionen abgeschlossen.

9.7 Reguläre Sprachen

9.7.1 Definition. Ein vollständiger deterministischer endlicher Automat $\mathscr{A}$ heißt *minimal* wenn jeder andere Automat, der dieselbe Sprache erkennt, mindestens soviel Zustände wie $\mathscr{A}$ hat.

9.7.2 Satz. *Zu jeder regulären Sprache gibt es, bis auf Isomorphie, genau einen diese erkennenden minimalen Automaten.*

9.7.3 Satz. *Alle folgenden Probleme sind für reguläre Sprachen entscheidbar:*

1. $L = \varnothing$;
2. L *ist endlich;*
3. $w \in L$;
4. $L_1 = L_2$;
5. $L_1 \subseteq L_2$.

Beweis. 1. Für die leere Sprache gibt es entweder gar keinen regulären Ausdruck, oder sie wird durch ein spezielles Symbol bezeichnet, dessen Vorkommen dann lediglich zu prüfen ist. Auch einem durch seinen Zustandsgraphen dargestellen Automaten kann man das leicht anmerken: er erkennt genau dann die leere Sprache, wenn es keinen Weg vom Anfangszustand zu einem Finalzustand gibt.

2. Ein regulärer Ausdruck beschreibt genau dann eine endliche Sprache, wenn keine Iteration vorkommt. Für einen Automaten müßte man prüfen, ob sein Zustandsgraph einen Zyklus enthält, von dem aus es eine Abzweigung zu einem Finalzustand gibt.

3. Festzustellen, ob ein Wort zu einer Sprache gehört oder nicht, ist gerade die Aufgabe, die ein erkennender Automat (optimal) beherrscht.

4. Es ist mitunter nicht so offensichtlich, ob zwei reguläre Ausdrücke dieselbe Sprache beschreiben. Man kann aber von beiden einen minimalen Automaten berechnen und feststellen, ob diese isomorph sind.

5. $L_1 \subseteq L_2$ gilt genau dann wenn $L_1 \cup L_2 = L_2$.

□

9.7.4 Lemma (Pumping Lemma). *Zu jeder regulären Spache L gibt es eine Zahl $n \in \mathbb{N}$, sodaß jedes Wort $z \in L$ mit $|z| \geq n$ derart in*

$$z = uvw$$

zerlegt werden kann, daß gilt:

- $|v| \geq 1$;
- $|uv| \leq n$;
- $uv^i w \in L$, *für alle* $i \in \mathbb{N}$.

Beweis. Laut Voraussetzung wird L von einem deterministischen endlichen Automaten erkannt. Wird nun ein Wort $z \in L$ eingegeben, dessen Länge n größer ist als die Anzahl der Zustände des Automaten, so muß zumindest ein Zustand z_2 mindestens zwei Mal angenommen werden. Entsprechend zerlegen wir $z = uvw$ derart, daß der Automat nach der Eingabe von u im Zustand q_2 ist und die Eingabe von v von q_2 wieder nach q_2 führt (w ist dann das Restwort). Dann gilt $|v| \geq 1$ und $|uv| \leq n$. Wird nun statt $z = uvw$ ein Wort der Form uv^iw eingegeben, so ändert sich nicht allzuviel, außer daß eben der Zyklus bei q_2 nun i Mal durchlaufen wird. □

Das Pumping Lemma kann man verwenden, um nachzuweisen, daß eine bestimmte Sprache nicht regulär ist. Dazu ist die negierte Variante praktisch:

9.7.5 Folgerung. *Sei L eine formale Spache L.*
Kann man zu jedem $n \in \mathbb{N}$ ein ein Wort $z \in L$ mit $|z| \geq n$ konstruieren derart, daß für jede Zerlegung

$$z = uvw$$

mit

- $|v| \geq 1$*;*
- $|uv| \leq n$*;*

ein i gefunden werden kann, sodaß $uv^iw \notin L$,
dann ist L nicht regulär.

9.8 Grammatiken

9.8.1 Definition. Eine *Grammatik* $\mathcal{G} = (T, N, S, \mathcal{P})$ besteht aus

- einem Alphabet Σ von *Terminalzeichen*;
- einem Alphabet N von *Nicht-Terminalzeichen*;
- einem Startsymbol $S \in N$;
- einer Menge von Produktionsregeln der Form

$$l \Rightarrow r,$$

mit $l \in (N \cup \Sigma)^* \setminus \Sigma^*$ und $r \in (N \cup \Sigma)^*$.

9.8.2 Definition. Sei $\mathcal{G} = (\Sigma, N, S, \mathcal{P})$ eine Grammatik, und $u, w \in (N \cup \Sigma)^*$. Dann ist w aus u *ableitbar* (Schreibweise: $u \twoheadrightarrow w$) wenn es Zerlegungen $u = ale$ und $w = are$ gibt, sodaß $l \Rightarrow r$ eine Produktionsregel der Grammatik ist.

Wie üblich, bezeichnet weiters $\twoheadrightarrow^*$ die reflexiv-transitive Hülle von $\twoheadrightarrow$.

9.8.3 Definition. Die von einer *Grammatik* $\mathcal{G} = (\Sigma, N, S, \mathcal{P})$ *erzeugte Sprache* ist definiert durch

$$L(\mathcal{G}) = \{u \in \Sigma^* \mid S \twoheadrightarrow^* u\}.$$

9.8.4 Beispiel. Gegeben sie die Sprache $\mathcal{G}_1 = \big(\{a,b\},\{S\},S,\mathcal{P}\big)$, wobei $\mathcal{P}$ aus folgenden Produktionen bestehe:

$$S \Rightarrow \epsilon,$$
$$S \Rightarrow SS,$$
$$S \Rightarrow aSb,$$
$$S \Rightarrow bSa.$$

Dann ist bespielsweise *abba* ein Wort der durch diese Grammatik erzeugten Sprache, denn es ist folgendermaßen aus S ableitbar:

$$S \Rightarrow SS \twoheadrightarrow aSbS \twoheadrightarrow aSbbSa \twoheadrightarrow^* abba.$$

Tatsächlich gilt:

$$L(\mathcal{G}_1) = \{w \in \{a,b\}^* \mid \#_a(w) = \#_b(w)\}.$$

Begründung: einerseits ist klar, daß keine Produktionsregel die Bedingung $\#_a(w) = \#_b(w)$ ändert, und umgekehrt läßt sich für jedes Wort, welches diese Bedingung erfüllt, eine Ableitung konstruieren. Sei z.B. $w = avb$, und gelte $\#_a(w) = \#_b(w)$. Dann gilt auch $\#_a(v) = \#_b(v)$, und daher (mit Induktion nach der Länge von w) $S \twoheadrightarrow^* v$. Mit $S \Rightarrow aSb$ folgt dann $S \twoheadrightarrow^* w$.

Ist eine Sprache durch eine Grammatik definiert, so können alle Wörter der Sprache effektiv aufgezählt werden. Ein passendes Programm muß nur, ausgehend vom Startsymbol, systematisch alle Möglichkeiten, Produktionsregeln anzuwenden, durchgehen, und alle dabei gefundenen Wörter von T^* ausgeben.

Ein Test, ob ein gegebenes Wort $w \in \Sigma^*$ zu der Sprache gehört, ist damit allerdings nicht möglich. Im positiven Fall läßt sich dies zwar feststellen (weil es irgendwann einmal in der Aufzählung vorkommt), aber im negativen Fall muß man mitunter unendlich lange warten (Semientscheidungsverfahren).

In vielen Fällen kann man sich auf gewisse Typen von Produktionsregeln beschränken. Die bekannteste Klassifizierung der Grammatiken is die *Chomsky-Hierarchie*.

9.8.5 Definition. Eine Grammatik ($\mathcal{G} = (\Sigma, N, S, \mathcal{P})$ heißt *rechtslinear*, falls alle Produktionsregel die Form

$$A \Rightarrow x \qquad \text{oder} \qquad A \Rightarrow xB$$

haben, mit $A, B \in N$, $x \in (N \cup \Sigma)^*$.

Analog definiert man *linkslineare* Grammatiken (mit $A \Rightarrow Bx$ statt $A \Rightarrow xB$).

9.8.6 Theorem. *Eine Sprache läßt sich genau dann durch eine rechtslineare Grammatik definieren, wenn sie regulär ist.*

Beweis. Wir können uns auf den Fall beschränken, daß alle Produktionsregeln der Form $A \Rightarrow xB$ ein x der Länge 1 verwenden (was gegebenenfalls durch Einführung zusätzlicher Nichtterminalsymbole erzwungen werden kann). Sei $\mathcal{G} = (\Sigma, N, S, \mathcal{P})$ eine rechtslineare Grammatik. Wir konstruieren eine Automaten, der $L(\mathcal{G})$ erkennt: Als Eingabemenge ist natürlich Σ zu wählen. Als

Zustandsmenge verwenden wir $N \cup \{\epsilon\}$, wobei $\epsilon \notin \Sigma \cup N$. Startzustand ist S, der einzige Finalzustand ist ϵ. Ein Zustandsübergang von einem Zustand A in einen Zustand $B \in N$ sei dann bei der Eingabe α genau dann möglich, wenn $A \Rightarrow \alpha B$ in $\mathcal{G}$ ist. Ferner sei ein Zustandsübergang von einem Zustand A in den Zustand ϵ bei der Eingabe α genau dann möglich, wenn $A \Rightarrow \alpha$ in $\mathcal{G}$ ist.

Umgekehrt kann man (auf dieselbe Weise) jedem Automaten eine rechtslineare Sprache zuordnen. □

9.8.7 Satz. *Eine Sprache L läßt sich genau dann durch eine linkslineare Grammatik definieren wenn ihre Spiegelung L^{r} durch eine rechtslineare Grammatik definierbar ist.*

Man beachte, daß die Klasse der regulären Sprachen unter Spiegelung abgeschlossen ist. Damit ergibt sich sofort:

9.8.8 Folgerung. *Eine Sprache ist genau dann durch eine linkslineare Grammatik definierbar wenn sie durch eine rechtslineare Grammatik definierbar ist.*

9.9 Kontexfreie Sprachen

9.9.1 Definition. Eine Grammatik $\mathcal{G} = (\Sigma, N, S, \mathcal{P})$ heißt *kontextfrei*, falls alle Produktionsregel die Form

$$A \Rightarrow r$$

haben, mit $A \in N$, $r \in (N \cup \Sigma)^*$. Eine Sprache heißt *kontextfrei* wenn sie durch eine kontextfreie Grammatik definiert werden kann.

Wird eine Sprache durch eine nicht-kontextfreie Grammatik definiert, so heißt dies noch lange nicht, daß sie nicht kontextfrei ist: es könnte ja eine andere, kontextfreie, Grammatik geben, welche dieselbe Sprache erzeugt.

9.9.2 Satz. *Jede reguläre Sprache ist kontextfrei.*

Beweis. Alle Produktionsregeln, die in einer rechtslinearen Grammatik erlaubt sind, sind auch in einer kontextfreien Grammatik erlaubt. □

9.9.3 Satz. *Zu jeder Ableitung in einer kontextfreien Grammatik gibt es einen* Ableitungsbaum *(auch:* Syntaxbaum*).*

9.9.4 Beispiel. Ableitungsbaum.

9.9.5 Definition. Eine Grammatik heißt *eindeutig*, wenn es zu jedem Wort der dadurch definierten Sprache genau einen Ableitungsbaum gibt.

9.9.6 Definition. Ein *Kellerautomat* ist ähnlich wie ein normaler (endlicher) Automat, kann aber zusätzlich auf einen Kellerspeicher (Stack, Stapel) zugreifen. Das heißt die Zustandsüberführung darf zusätzlich vom obersten Zeichen im Kellerspeicher abhängen, und sie darf auch, als Nebeneffekt, ein Zeichen im Kellerspeicher ablegen oder davon entfernen. Ein Kellerautomat *erkennt* ein Eingabewort, wenn er sich nach dessen Abarbeitung in einem Finalzustand befindet *und* der Keller leer ist.

9.9.7 Satz. *Kellerautomaten erkennen genau die kontextfreien Sprachen.*

9.9.8 Beispiel. Kellerautomat zu kontextfreier Grammatik.

Um nachzuweisen, daß eine Sprache nicht kontextfrei ist, genügt es natürlich nicht, wenn sie durch eine nicht-kontextfreie Grammtik gegeben ist. Man kann zu diesem Zweck wieder ein Pumping Lemma verwenden.

9.9.9 Theorem. *Für jede kontextfreie Sprache ist entscheidbar, ob sie*

- *ein bestimmtes Wort enthält;*
- *leer ist;*
- *endlich ist.*

Nicht entscheibar dagegen ist, ob zwei kontextfreie Sprachen gleich sind oder ob die eine die andere enthält.

9.9.10 Theorem. *Die Klasse der kontextfreien Sprachen ist abgeschlossen gegenüber Verkettung, Iteration und Vereinigung, nicht aber gegenüber Komplement und Durchschnitt.*

9.9.11 Lemma (Pumping Lemma). *Zu jeder kontextfreien Spache L gibt es eine Zahl $n \in \mathbb{N}$, sodaß jedes Wort $z \in L$ mit $|z| > n$ derart in*

$$z = uvwxy$$

zerlegt werden kann, daß gilt:

- $|vx| > 0$*;*
- $|vwx| \leq n$*;*
- $uv^iwx^iy \in L$*, für alle* $i \in \mathbb{N}$*.*

9.9.12 Beispiel. Die Sprache

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

ist nicht kontextfrei.

Für jedes $n \in \mathbb{N}$ wählen wir

$$z = a^n b^n c^n,$$

welches offensichtlich in L liegt und $|z| > 0$ erfüllt.

Sei nun eine Zerlegung

$$z = uvwxy$$

gegeben, welche $|vx| > 0$ und $|vwx| \leq n$ erfüllt. Die erste Bedingung ergibt, daß v und x nicht beide leer sein können, und aus der zweiten erhalten wir, daß in vwx nicht alle drei Buchstaben vorkommen können. Wir müssen dann ein i finden, sodaß $uv^iwx^iy \notin L$. Wir wählen $i = 2$. Tatsächlich kommt der Buchstabe, der in vwx nicht vorkommt, in uv^2wx^2y genauso oft vor wie in $uvwxy$, was aber für zumindest einen der anderen Buchstaben nicht zutrifft. Daher ist $uv^2wx^2y \notin L$, woraus folgt, daß L nicht kontextfrei sein kann, weil das Pumping Lemma verletzt ist.

9.10 Kontextsensitive Sprachen

9.10.1 Definition. Eine Grammatik $\mathcal{G} = (\Sigma, N, S, \mathcal{P})$ heißt *kontextsensitiv*, falls alle Produktionsregel die Form

$$uAv \Rightarrow urv$$

haben, mit $A \in N$, $r, u, v \in (N \cup \Sigma)^*$, $r \neq \epsilon$. Eine Sprache heißt *kontextsensitiv* wenn sie durch eine kontextsensitive Grammatik definiert werden kann.

Wird eine Sprache durch eine nicht-kontextsensitive Grammatik definiert, so heißt dies noch lange nicht, daß sie nicht kontextsensitiv ist: es könnte ja eine andere, kontextsensitive, Grammatik geben, welche dieselbe Sprache erzeugt.

9.10.2 Satz. *Jede kontextfreie Sprache ist kontextsensitiv.*

Beweis. Die meisten Produktionsregeln, die in einer kontextfreien Grammatik erlaubt sind, sind auch in einer kontextsensitiven Grammatik erlaubt. Ausnahme: Regeln der Form $A \Rightarrow \epsilon$. Es läßt sich aber zeigen (nicht ganz so leicht), daß man auf derartige Regeln verzichten kann. □

Es läßt sich sogar zeigen, daß jede *monotone* Grammatik (bei allen Regeln ist die rechte Seite leer oder länger als die linke) eine kontextsensitive Sprache beschreibt.

Eine *linear beschränkte Turingmaschine* ist eine Turingmaschine (bzw. Rechenmaschine, Computer), die mit einem Hilfsspeicher auskommt, dessen Größe durch eine konstantes Vielfaches der Länge des Eingabewortes beschränkt werden kann.

9.10.3 Theorem. *Eine Sprache ist genau dann kontextsensitiv, wenn sie durch eine linear beschränkte Turingmaschine erkannt wird.*

9.10.4 Theorem. *Für jede kontextsensitive Sprache ist entscheidbar, ob sie ein bestimmtes Wort enthält, nicht aber ob sie leer oder endlich ist. Nicht entscheibar ist auch, ob zwei kontextsensitive Sprachen gleich sind oder ob die eine die andere enthält.*

9.10.5 Theorem. *Die Klasse der kontextsensitiven Sprachen ist abgeschlossen gegenüber Verkettung, Iteration sowie allen boolschen Mengenoperationen.*

Im Vergleich zu den kontextfreien Sprachen gewinnen wir damit wieder alle Abgeschlossenheitseigenschaften (wie bei den regulären Sprachen) zurück, müssen dafür aber zwei Entscheidbarkeitseigenschaften opfern, nicht aber die wichtigste.

9.11 Turingmaschine

Ein Automat hat außer einer endlichen Mengen von Zuständen Zugriff auf ein Eingabeband, welches sequentiell gelesen wird, und, wenn es ein Vollautomat ist, auf ein Ausgabeband, welches sequentiell beschrieben wird. Es ist nicht möglich, diese Bänder als Hilfsspeicher zu verwenden. Da der interne Speicher eines Automaten daher stets endlich ist, kann er nur reguläre Sprachen erkennen. Eine Verallgemeinerung stellen die Kellerautomaten dar, welche zusätzlich auf

einen Kellerspeicher zugreifen können, und daher auch kontextfreie Sprachen erkennen können.

Eine deutliche Verallgemeinerung ergibt sich, wenn das Eingabeband auch beschrieben und beliebig hin- und herbewegt werden kann. Man nennt dies dann eine *Turingmaschine*. Sie entspricht in etwa dem, wie man auch auf Papier mit Bleistift und Radiergummi mit Symbolen hantiert, d.h. rechnet. Das Band der Turingmaschine ist theoretisch unendlich lange; in endlicher Zeit kann aber nur ein endlicher Teil davon beschrieben werden. Dies entspricht einem unerschöpflichen Vorrat an Schmierpapier.

Der Folgezustand einer Turingmaschine ergibt sich aus dem momentanen Zustand und dem Zeichen an der aktuellen Leseposition des Bandes. Bei jeder Zustandsänderung wird darüberhinaus ein geeignetes Zeichen auf das Band geschrieben und dieses eventuell nach vor oder zurück bewegt. Gelegentlich ist es bequemer, auch mehrere Bänder zuzulassen. Außerdem gibt es auch bei den Turingmaschinen, so wie bei den Automaten, deterministische und nichtdeterministische Varianten.

Eine deterministische Turingmaschine entspricht daher grob der Arbeitsweise eines gängigen Computers (mit beliebig erweiterbarem Speicher), während eine nicht-deterministische Turingmaschine einem Parallelrechner mit einem unbeschränkten Vorrat an Prozessoren entspricht.

Die Turingmaschine wurde 1936 von Alan Turing eingeführt, um Berechenbarkeit und Entscheidbarkeit zu studieren. In der Folge wurde zahlreiche weitere Modelle für Rechenmaschinen entwickelt. Es stellte sich heraus, daß all diese Machinen dieselben Probleme bewältigen können, also gewissermaßen äquivalent sind. Man kann sich daher unter einer Turingmaschine genausogut ein Programm in einer gängigen Programmiersprache oder einen konkreten Computer vorstellen, oder auch einen Menschen, der mit Symbolen hantiert. Es liegt daher nahe, festzulegen, daß ein Problem genau dann berechenbar oder entscheidbar ist, wenn dies eine Turingmaschine zu leisten vermag. Diese Annahme heißt auch *Churchsche These*.

9.12 Rekursiv und Rekursiv aufzählbar

Sei Σ ein endliches Alphabet.

9.12.1 Definition. Eine Funktion $\Sigma^* \to \Sigma^*$ heißt *rekursiv*, wenn sie durch eine Turingmaschine berechenbar ist.

9.12.2 Definition. Eine Teilmenge von Σ^* (Sprache) heißt *rekursiv*, wenn ihre Indikatorfunktion rekursiv ist.

Wenn eine Funktion von einer Turingmaschine berechnet wird, so bedeutet dies insbesondere, daß sie für jede mögliche Eingabe irgendwann einmal stehen bleibt (in einem finalen Zustand landet). Im allgemeinen kann es aber passieren, daß sie unentwegt fortläuft, ohne je ein Ergebnis zu liefern.

9.12.3 Definition. Eine Sprache $A \subseteq \Sigma^*$ heißt rekursiv aufzählbar falls sie von einer Turingmaschine *erkannt* wird, d.h. wenn es eine Turingmaschine gibt, welche genau dann irgendwann einmal stehen bleibt, wenn ein $w \in A$ eingegeben wird.

9.12.4 Satz. *Jede durch eine Grammatik definierte Sprache ist rekursiv aufzählbar.*

9.12.5 Satz. *Jede kontextsensitive Sprache ist rekursiv.*

9.13 Abschlußeigenschaften

Operation	regulär	kontextfrei	kontextsensitiv	rekursiv	rekursiv aufzäh
Verkettung	ja	ja	ja	ja	ja
Iteration	ja	ja	ja	ja	ja
Vereinigung	ja	ja	ja	ja	ja
Durchschnitt	ja	nein	ja	ja	ja
Komplement	ja	nein	ja	ja	nein

9.14 Entscheidbarkeitseigenschaften

Problem	regulär	kontextfrei	kontextsensitiv	rekursiv	rekursiv aufzählba
$w \in L$	ja	ja	ja	ja	nein
$L = \varnothing$	ja	ja	nein	nein	nein
L endlich	ja	ja	nein	nein	nein
$L_1 = L_2$	ja	nein	nein	nein	nein
$L_1 \subseteq L_2$	ja	nein	nein	nein	nein

9.15 Berechenbarkeit

Wir haben die logischen Operatoren (Konjunktion, Disjunktion, Implikation, Allquantor, Existenzquantor) mittels Introduktions- und Eliminationsregeln definiert. Ebenso $\bot$, die Aussage, die stets falsch ist, und $\top$, und $\neg P$ steht als Abkürzung für $P \implies \bot$. Dieses logische System heißt konstruktive (genauer: intuitionistische Logik).

Ganz analog dazu haben wir Mengen, insbesondere die natürliche Zahlen, eingeführt. Wenn mittels dieser Regeln eine Funktion $f\colon A \to B$ definiert wird, so ist sichergestellt, daß zu jedem $\alpha \in A$ effektiv ein Term für $f(x)$ angegeben werden kann.

Durch Hinzunahme der Regel *Tertium non datur*, dem Prinzip vom ausgeschlossenen Dritten, ergibt sich die *klassische Logik*, welche der Großteil der mathematischen Literatur des 20. Jahrhunderts verwendet.

Die klassische Logik ist irgendwie einfacher, denn sie hat nur 2 Wahrheitswerte, während die konstruktive Logik unendlich viele Wahrheitswerte unterscheidet, die nochdazu sehr kompliziert strukturiert sind. Außerdem gibt es in der klassischen Logik mehr Möglichkeiten, Sätze zu beweisen, und man kann damit auch Sätze beweisen, die konstruktiv nicht bewiesen werden könnten.

Ein Nachteil der klassischen Logik ist dagegen, daß damit die Existenz von Funktionen bewiesen werden kann, die nicht einmal theoretisch berechnet werden können, also gar nicht „funktionieren“. Dies ist insbesondere in der Computerwissenschaft recht lästig, da hier nur die berechenbaren Funktionen interessant sind, und zu deren Definition man Maschinenmodelle benötigt, von denen

nicht unbedingt klar ist, ob damit wirkich der intuitive Begriff *berechenbar* erfaßt wird.

Die Annahme, daß eine Funktion genau dann berechenbar ist, wenn sie auf einer Turing-Maschine berechenbar ist, heißt *Church-Turing-These*. Sie wird dadurch gerechtfertigt, daß die Funktionsweise heutiger Computer im wesentlichen der Turingmachine entspricht, sowie, daß zahlreiche weitere Methoden, den Berechenbarkeitsbegriff festzulegen, dazu äquivalent sind *rekursive Funktionen*.

Allerdings gibt es dabei auch Probleme: Einerseits ist der Begriff *rekursiv* einheitlich nur Funktionen von eimem Typ wie $\Sigma^* \to \Sigma^*$ oder $\mathbb{N} \to \mathbb{N}$ definiert, nicht aber z.B. für Funktionen vom komplizierterem Typ, wie $\mathbb{R} \to \mathbb{R}$.

Andererseits betrachten wir die Funktion $p\colon \mathbb{N} \to \mathbb{N}$

$$p(n) = \begin{cases} 0 & \text{falls es unendlich viele Primzahlzwillinge gibt} \\ 1 & \text{falls es nur endlich viele Primzahlzwillinge gibt.} \end{cases}$$

Mittels Prinzip vom ausgeschlossenen Dritten erkennen wir sofort, daß diese Funktion konstant ist. Damit ist sie auch rekursiv. Andererseits weiß niemand, ob es unendlich viele Primzahlzwillinge gibt oder nicht. (Und wenn diese Frage doch irgendwann einmal geklärt werden sollte, nimmt man einfach irgendein anderes ungeklärtes Problem.) Daher haben wir keine Möglichkeit zur Verfügung, auch nur einen Funktionswert, etwa $p(1)$, zu berechenen, nichteinmal, wenn uns beliebig viel Zeit und Speicherplatz zur Verfügung stünde. Schlimmer noch, es könnte sogar sein, daß die Frage, ob es unendlich viele Primzahlzwillinge gibt oder nicht, tatsächlich unentscheidbar ist (jedenfalls weiß man, daß es solche unentscheibare Aussagen über die natürlichen Zahlen gibt).

Bis auf diese solche Fragen im Grenzbereich stimmen aber die rekursiven Funktionen mit denjenigen, deren Existenz konstruktiv bewiesen werden kann, überein, und beschreiben sehr gut den intuitiven Berechenbarkeitsbegriff.

9.16 Entscheidbarkeit

Das Prinzip vom ausgeschlossenen Dritten liefert definitiv einen Widerspruch, wenn es eine Aussage P gibt, sodaß

$$P \iff \neg P,$$

also eine Aussage, die genau dann wahr ist, wenn sie falsch ist. Tatsächlich läßt die deutsche Sprache derartige Aussagen zu, z.B.:

$$P = \text{„Dieser Satz ist falsch“}.$$

Wenn P wahr ist, dann heißt das gerade, daß P falsch ist, und wenn P falsch ist, dann stimmt der Datz offensichtlich. Gilt nun das Prinzip vom ausgeschlossenen Dritten, so hat man damit einen Widerspruch, und damit ist jeder Satz sowohl wahr als auch falsch (*ex falsum quodlibet*), wodurch das ganze logische System sinnlos wird.

Damit ist zuerst einmal klar, daß man bei natürlichen Sprachen die Regeln für klassische Logik nur mit Bedacht anwenden darf. Für formale Sprachen dagegen kann man derartige selbstbezügliche Sätze ausschließen. Sobald man allerdings die natürlichen Zahlen verwendet, hat man auch das Rekursionsprinzip,

welches ja auch eine Art Selbstbezüglichkeit ausdrückt. Tatsächlich gelingt es einen Satz über die natürlichen Zahlen zu konstruieren, der dem obigen Satz P entspricht. Dies ist ein bemerkenswertes klassisches Resultat der Logik:

9.16.1 Theorem (Gödel'scher Unvollständigkeitssatz). *Jedes formale logische System, welches ausdrucksstark genug ist, um darin die natürlichen Zahlen zu beschreiben, ist entweder widersprüchlich oder beinhaltet unbeweisbare Sätze, von denen auch die Negation nicht beweisbar ist.*

Das Prinzip vom ausgeschlossenen Dritten läßt sich daher nur dadurch retten, daß man klar zwischen *beweisbaren* und *wahren* Sätzen unterscheidet. Dabei ergibt sich allerdings das praktische Problem, daß es unmöglich ist, die Klasse der wahren Sätze über die natürlichen Zahlen irgendwie formal zu definieren (sie sind nicht rekursiv aufzählbar). Für die Praxis relevant sind daher in jedem Fall nur die beweisbaren Sätze.

Eine Variante dieses Satzes besagt, daß es unmöglich ist, die Konsistenz eines hinreichend komplexen logischen Systems zu beweisen.

9.17 Komplexitätsklassen

Um die Komplexität von Algorithmen festzustellen, zählt man, wieviele Schritte dafür auf einer bestimmten Maschine (etwa einer Turingmaschine) notwendig sind Diese Zählung ist freilich sehr vom verwendeten Maschinenmodell ab und ist außerdem unnötig kompliziert. Meist versucht man, die Komplexität durch eine von der Länge des Inputs abhängige Funktion zu beschränken, wobei konstante Faktoren unberücksichtigt bleiben (O-Notation), womit, außer der Vereinfachung, auch eine gewisse Maschinenunabhängigkiet erreicht wird.

Zu O-Notation: http://en.wikipedia.org/wiki/Big_O_notation

9.17.1 Definition. Ein Algorithmus ist *polynomial*, wenn es einen (fixen!) Exponenten k gibt, sodaß die Komplexität bei einem Input der Länge n durch $O(n^k)$ beschränkt werden kann.

9.17.2 Beispiel.

Die Addition von höchstens n-stelligen Binär- oder Dezimalzahlen ist (mit dem üblichen Algorithmus) durch $O(n)$ beschränkt, also insbesondere polynomial (mit $k = 1$, er ist daher sogar *linear*).

Der übliche Algorithmus für die Multiplikation benötigt $O(n^2)$ Schritte, ist also ebenfalls polynomial, aber nicht linear. (Es gibt aber auch schnellere Algorithmen, die sich bei sehr großen Zahlen auszahlen; der Rekord liegt derzeit bei $O(n \log n \log\log n)$, was nur geringfügig mehr als linear ist).

9.17.3 Definition.

- Ein Problem ist in der *Klasse P*, wenn es mit einer deterministischen Turingmaschine in polynomialer Zeit berechnet werden kann.
- Ein Problem ist in der *Klasse NP*, wenn es auf einer nicht-deterministischen Turingmaschine in polynomialer Zeit berechnet werden kann. Oder: wenn dessen Lösung mit einer deterministischen Turingmaschine in polynomialer Zeit überprüft werden kann.

- Ein Problem ist in der *Klasse EXPSPACE*, wenn es mit einer Turingmaschine mit einem höchstens exponentiell ansteigenden Speicher (wieder abhängig von der Länge des Inputs) gelöst werden kann.
- Ein Problem heißt *primitiv rekursiv* (*Klasse PR*), wenn sie mit primitiver Rekursion (das heißt vereinfacht: nur mit for-Schleifen, aber ohne while-Schleifen) gelöst werden kann.
- Ein Problem ist rekursiv (mit while-Schleifen, aber terminierend) wird mit R bezeichnet.
- Die *Klasse RE* besteht aus den Problemen, für die eine Turingmaschine in endlicher Zeit JA sagen kann, aber statt der Antwort NEIN möglicherweise nie terminiert.

Natürlich gibt es dazwischen noch mehrere Abstufungen und Verfeinerungen, vgl. etwa
http://en.wikipedia.org/wiki/List_of_complexity_classes, oder noch genauer http://qwiki.caltech.edu/wiki/Complexity_Zoo.

9.17.4 Satz. *Es gilt:*

$$P \subseteq NP \subset EXPSPACE \subset PR \subset R \subset RE$$

Es ist ein bekanntes offenes Problem, ob $P = NP$ gilt.

9.17.5 Definition. Ein Problem ist *NP-vollständig* (*NP-complete*) wenn es in der Klasse NP ist und jedes andere NP-Problem in polynomialer Zeit darauf zurückgeführt werden kann.

9.17.6 Beispiel. Die üblichen arithmetischen Operationen, insbesondere das Lösen von linearen Gleichungssymstem und linearen (nicht-diskreten) Optimierungsproblemen), sowie die „einfacheren“ graphentheoretischen Probleme (Eulerscher Weg, kürzester Weg, maximaler Durchfluß, Planarität, 2-Färbbarkeit) sind in der Klasse P. Auch das Erkennen kontexfreier Sprachen ist in der Klasse P.

Viele diskrete Optimierungsaufgaben und Suchprobleme sind in der NP-vollständig, z.B. Graphen-Isomorphie, Hamiltonscher Weg in einem Graphen, Travelling-Salesman-Problem, 3-Färbbarkeit.

Das Lösen von algebraischen Gleichungssystemen über den rationalen Zahlen ist in EXPSPACE.

Die Preßburger Arithmetik (eine Theorie der natürlichen Zahlen ohne Rekursion) oder die Theorie der formal rellen Körper (eine Theorie der reellen Zahlen ohne Grenzwerte) sind noch schwieriger, lassen sich mit primitiver Rekursion entscheiden.

Die Ackermann-Funktion ist rekursiv, aber nicht primitiv rekursiv.

Die Klasse der (in einem formalen System) beweisbaren Sätze ist rekursiv aufzählbar, aber im allgemeinen nicht rekursiv. Ebenso die Menge der pariell rekursiven Funktionen, oder die Menge der rekursiv aufzählbaren Probleme.

Die Menge der wahren Sätze über die natürlichen Zahlen ist nicht rekursiv aufzählbar, ebenso wie die Menge der rekursiven Funktionen oder der Probleme der Klasse R.