Automaten, Formale Sprachen und Berechenbarkeit I

Skript zur Vorlesung im WS 2001/02

an der TU München

Ekkart Kindler Steffen Manthey

Version: 1.30 vom 12. Mai 2002

Redaktioneller Hinweis: Es gibt ein paar Dinge, die noch Vereinheitlicht werden müssen.

- *Symbole für Zeichen des Alphabets: sollten nicht fett, gesetzt werden.*
- *Alte und neue Rechschreibung (z.B. daß vs. dass)*

Teil A

Formalia und Einführung

Vorwort

Dieses Skript ist im Anschluß an die gleichnamige Vorlesung entstanden, die ich im WS 2001/02 an der Technischen Universität München gehalten habe. Eigentlich hatte ich nicht die Absicht, eine „geTeXte“ Version meines Vorlesungsskriptes zu erzeugen, da sich meines Erachtens ein gutes Skript über mehrere Semester entwickeln muß. So gesehen ist dieses Schriftstück also noch gar kein Skript, sondern nur eine Vorversion.
Da sich jedoch Herr Manthey die Mühe gemacht hat, seine Mitschrift zu „TeXen“, habe ich nun also doch begonnen, aus dieser Mitschrift ein Skript zu erstellen, das sich hoffentlich über die Jahre noch etwas verbessern wird. Noch sind einige „saloppe“ Bemerkungen enthalten, die im Rahmen einer Vorlesung sicher gut ankommen, aber in einem Skript etwas merkwürdig anmuten. Auch fehlen an vielen Stellen noch ausführlichere Erklärungen und Kommentare (vor allem am Ende), die in der Vorlesung nur mündlich gegeben wurden oder sogar nur durch Gestik und Mimik vermittelt wurden. Leider fehlen auch noch einige graphische Darstellungen zur Veranschaulichung bestimmter Konzepte oder Beweise.
Ich hoffe jedoch, daß bereits diese Vorversion des Skriptes eine gute Ergänzung zu den gescannten handschriftlichen Unterlagen im WWW darstellt:

`http://wwwbrauer.informatik.tu-muenchen.de/lehre/fospr/WS0102/`

Ich bedanke mich bei den Studierenden, die sich aktiv und motiviert an der Vorlesung und der Übung beteiligt haben. Sie haben mich auf Fehler hingewiesen, mich durch Rückfragen dazu gebracht, über schlüssigere Erklärungen nachzudenken und mich motiviert, dieses Skript überhaupt zu erzeugen.
Ganz besonderer Dank gilt Herrn Manthey, der viel Mühe aufgewendet hat, in der Vorlesung mitzutippen, und so die „TeX“nische Grundlagen zu diesem Skript gelegt hat. So konnte ich mich überwiegend auf die inhaltliche Überarbeitung und Ergänzung des Skriptes konzentrieren.

München, im April 2002,
Ekkart Kindler

Inhaltsverzeichnis

Kapitel 1

Einführung

1 Motivation

Frage: Warum sollte sich ein “praktischer Informatiker” oder “Softwaretechniker” diese Theorie-Vorlesung anhören.

Antwort 1: So theoretisch ist das Thema nicht (wird benötigt für: Compilerbau, Sprachentwurf, ...).

Antwort 2: Für eine weitere Antwort gehen wir zuerst der Frage nach: Was ist Informatik?

Informatik beschäftigt sich mit der mechanischen Verarbeitung, Übermittlung, Speicherung und Wiedergewinnung von Informationen und mit der Konstruktion von Systemen, die diesem Zweck dienen.

Was wiederum eine Information ist, wird an dieser Stelle nicht behandelt, da dies hier zu weit führen würde. Es werden sowieso nur die Repräsentationen der Informationen verarbeitet und nicht die Information selbst. Diese Informationen werden durch Zeichenketten über einem Alphabet repräsentiert.

Eine *(Formale) Sprache* ist aber nichts anderes als eine Menge von Zeichenreihen (Wörtern) über einen Alphabet.

- *Beschreibung von Klassen von Informationen*
- *Beschreibung der Struktur von Informationen*

Ein *Automat* ist ist ein abstraktes Modell eines mechanischen Systems zur Verarbeitung von Informationen (Zeichenreihen).

- *Abstraktion von technischen Details*
- *Konzentration auf das Wesentliche*
- *Klassifikation von Automatentypen und Untersuchung von Beziehungen*

Die *Berechenbarkeitstheorie* untersucht die Grenzen mechanischer Systeme zur Verarbeitung von Informationen (Zeichenreihen) und zur Berechnung von Funktionen.

In dieser Vorlesung werden also grundlegende Konzepte, Modelle und Beweistechniken vorgestellt, die der Verarbeitung von Information, der Berechnung von Funktionen dienen und die sich zur Untersuchung und dem Verständnis der ihnen zugrundeliegenden Prinzipien als zweckmäßig erweisen haben.

Diese sollen sich im Unterbewußtsein eines Informatikers fest verankern, auch wenn er sie nicht explizit benutzt (niemand soll mit der Turing-Maschine programmieren).

Antwort 3: Die in der Vorlesung vorgestellten Ideen, Denkweisen und Beweise schulen das "informatische" Denken.

- Bilden von Abstraktionen / Modellen

- Erkennen von Zusammenhängen

- Führen von Beweisen auf verschiedenen Abstraktionsebenen

2 Überblick

Zunächst geben wir einen Überblick darüber, was uns in der Vorlesung erwartet. Abbildung 1.1 zeigt die in der Vorlesung betrachteten Sprachklassen und die zugehörigen Automatentypen. Dies ist eine grobe Landkarte des Gebiets, auf dem wir uns in der Vorlesung bewegen. Diese Landkarte nennt sich *Chomsky-Hierarchie*.

Sprachklasse	Grammatik	Automat	Beispiel	andere Charakterisierung
rekursive aufzählbare Sprachen	Typ 0	NTM = DTM	"Halteproblem"	RAM, μ-Rekursion $\vee$ primitive Rekursion
rekursive Sprachen				
kontextsensitive Sprachen	Typ 1	LBA	$a^n b^n c^n$	
kontextfreie Sprachen	Typ 2	NKA	korrekt geklammerte Ausdrücke	
deterministische kontextfreie Sprachen	LR(k), LL(k)	DKA	$a^n b^n$	
lineare Sprachen				
reguläre Sprachen	Typ 3	NEA = DEA	$a^* b^*$	rationale (reguläre) Ausdrücke, erkennbare Mengen, ...

DEA *Deterministischer endlicher Automat*
NEA *Nicht-deterministischer endlicher Automat*
DKA *Deterministischer Kellerautomat*
NKA *Nicht-deterministischer Kellerautomat*
LBA *Linear beschränkter Automat*
(Turing-Maschine mit einem linear beschränktem Band)
DTM *Deterministischer Turing-Maschine*
NTM *Nicht-deterministischer Turing-Maschine*

Tabelle 1.1. Grobe Übersicht über die wichtigsten Sprachklassen: Die Chomsky-Hierarchie

Achtung: Diese Landkarte ist hier noch sehr ungenau und streng genommen in einigen Punkten sogar falsch. Die Welt ist eben nicht so linear wie hier dargestellt. Die genauen Beziehungen zwischen den Sprachklassen werden wir später noch angeben.

In der Vorlesung werden wir also verschiedene Sprachklassen definieren und untersuchen. Typische Fragestellungen sind dann:

1. Formale Sprachen und Automaten
   - Äquivalenz und Inklusion von Sprachklassen
   - Äquivalenz von Automatentypen (zum Beispiel: gilt DLBA = NLBA ?)
   - Entscheidungsverfahren / Unentscheibarkeit
     · Ist ein Wort in einer Sprache enthalten ?
     · Ist eine Sprache endlich ?
     · Ist die Sprache leer ?
   - Abschlußeigenschaften
     · Ist der Durchschnitt / die Vereinigung zweier Sprachen einer bestimmten Klasse wieder aus dieser Klasse?
     · Ist das Komplement einer Sprache einer bestimmten Klasse wieder aus dieser Klasse?
2. Berechenbarkeit
   - Äquivalenz der verschiedenen Berechnungsmodelle (Church'sche These)
   - Grenzen des Berechenbaren

Bereits in Abschnitt 3.3 wird uns eine sehr einfache Überlegung eine traurige Wahrheit lehren:
Es gibt wesentlich mehr Funktionen, die nicht berechenbar sind, als Funktionen, die berechenbar sind.

3 Grundbegriffe

An dieser Stelle werden die mathematischen Grundbegriffe eingeführt, die wir in der Vorlesung voraussetzen. Hier werden jedoch nur die wichtigsten Begriffe erwähnt; weitere Begriffe werden bei Bedarf eingeführt. Eigentlich sollten alle hier aufgeführten Begriffe bereits bekannt sein. Dieses Kapitel dient also im wesentlichen dazu, Sie mit der in der Vorlesung benutzten Terminologie und Notation vertraut zu machen.

3.1 Wörter und Sprachen

Ein *Alphabet* ist eine endliche Menge von *Zeichen* (*Symbolen*). Ein Beispiel ist das Alphabet $\Sigma = \{a, b, c\}$. Eine endliche Sequenz $a_1 a_2 \dots a_n$ mit $a_i \in \Sigma$ für $i \in \{1, \dots, n\}$ nennen wir *Wort* (*Zeichenreihe, Zeichenkette*) über dem Alphabet Σ. Die Menge aller Wörter über Σ bezeichnen wir mit Σ^*. Als Bezeichner für Wörter benutzen wir typischerweise die Buchstaben u, v, w (da a, b, c, ... bereits für die Zeichen des Alphabets reserviert sind). Die leere Sequenz bezeichnen wir mit ε; wir nennen ε das *leeres Wort*. Es gilt:

$$\begin{aligned} \varnothing^* &= \{\varepsilon\} \\ \{a\}^* &= \{\varepsilon, a, aa, aaa, \dots\} \end{aligned}$$

Für zwei Wörter $v = a_1 a_2 \dots a_n \in \Sigma^*$ und $w = b_1 b_2 \dots b_m \in \Sigma^*$ definieren wir die *Konkatenation* $v \circ w$ (kurz: vw) von v und w durch:

$$v \circ w = vw = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$$

Für Wort $w = a_1 a_2 \dots a_n$ heißt n auch die *Länge des Wortes* w und wir schreiben $|w| = n$. Die Länge eines Wortes kann man induktiv wie folgt definieren:

$$\begin{aligned} |\varepsilon| &= 0 \\ |a| &= 1 \qquad \text{für } a \in \Sigma \\ |v \circ w| &= |v| + |w| \end{aligned}$$

Ein Wort $u \in \Sigma^*$ heißt *Präfix* eines Wortes $v \in \Sigma^*$, wenn ein Wort $w \in \Sigma^*$ mit $u \circ w = v$ existiert. Ein Wort $u \in \Sigma^*$ heißt *Suffix* eines Wortes $v \in \Sigma^*$, wenn ein Wort $w \in \Sigma^*$ mit $w \circ u = v$ existiert.
Eine Teilmenge $L \subseteq \Sigma^*$ nennen wir *Sprache* über Σ. Eine Wort $w \in L$ nennen wir dann auch Wort der Sprache L.
Die *Konkatenation* zweier Sprachen $L_1, L_2 \subseteq \Sigma^*$ ist definiert durch:

$$L_1 \circ L_2 = L_1 L_2 = \{vw \mid v \in L_1, w \in L_2\}$$

Die *Kleene'sche Hülle* L^* einer Sprache L ist definiert durch:

$$\begin{aligned} L^0 &= \{\varepsilon\} \\ L^{i+1} &= LL^i \qquad \text{für } i \geq 1 \\ L^* &= \bigcup_{i \in \mathbb{N}} L_i \\ L^+ &= L^* \setminus \{\varepsilon\} \end{aligned}$$

Achtung der Stern-Operator hat in der Kleene'schen Hülle eine etwas andere Bedeutung als in Σ^*. Der Stern-Operator auf Sprachen wird auch Kleene-Stern *(bzw.* Kleene star*) genannt.*

3.2 Äquivalenzrelationen

Eine Teilmenge $R \subseteq A \times A$ wird *binäre Relation* über A genannt. Für $(a, b) \in R$ wird im allgemeinen die Schreibweise $a \; R \; b$ verwendet (oft auch $a \to b$ oder $a \Rightarrow b$). R heißt:

- *reflexiv*, wenn für alle $a \in A$ gilt $a \; R \; a$
- *symmetrisch*, wenn für alle $a, b \in A$ mit $a \; R \; b$ auch $b \; R \; a$ gilt.
- *transitiv*, wenn für alle $a, b, c \in A$ mit $a \; R \; b$ und $b \; R \; c$ auch $a \; R \; c$ gilt.

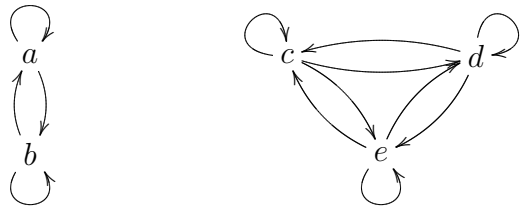


Abbildung 1.1. Eine Äquivalenzrelation mit zwei Äquivalenzklassen $[a] = \{a, b\}$ und $[c] = \{c, d, e\}$.

Eine Relation R heißt *Äquivalenz(relation)*, wenn R reflexiv, transitiv und symmetrisch ist.
Eine Äquivalenz R zerlegt eine Menge A in disjunkte *Äquivalenzklassen*: $[a]_R = \{b \in A \mid a\ R\ b\}$. Für die Äquivalenzklassen gilt:

$$\begin{array}{lcll} [a]_R & = & [b]_R & \text{gdw. } a\ R\ b \\ [a]_R \cap [b]_R & = & \varnothing & \text{gdw. } \neg a\ R\ b \\ A & = & \bigcup\limits_{a \in A} [a]_R & \end{array}$$

Die Anzahl der Äquivalenzklassen von R heißt *Index* von R: $|\{[a]_R \mid a \in A\}|$. Abbildung 1.1 zeigt eine Äquivalenzrelation über $A = \{a, b, c, d, e\}$ mit zwei Äquivalenzklassen; sie hat also den Index 2.
Die *transitive Hülle* R^+ (bzw. $\rightarrow^+$) einer (beliebigen) Relation R ist die kleinste Relation, die transitiv ist und R enthält. Man kann die transitive Hülle einer beliebigen Relation „konstruieren“, indem man so lange Kanten in R einfügt, bis die Transitivitätsbedingung erfüllt ist.
Die *reflexiv-transitive Hülle* R^* (bzw. $\rightarrow^*$) einer Relation R ist die kleinste Relation, die reflexiv und transitiv ist und R enthält. Man kann die reflexiv-transitive Hülle einer Relation konstruieren, indem man zusätzlich zur transitiven Hülle der Relation an jedem Knoten die Kante $\circlearrowleft$ einfügt.

3.3 Abzählbarkeit

Motivation

Wieviele Funktionen $g : \mathbb{N} \rightarrow \{0, 1\}$ gibt es?	unendlich viele
	$\vee$
Wieviele Programme gibt es?	unendlich viele
	$\|\|$
Wieviele natürliche Zahlen gibt es?	unendlich viele

Es gibt verschiedene Stufen von Unendlichkeit. Wir benötigen also ein Handwerkszeug, um die verschiedenen Stufen von Unendlichkeit unterscheiden zu können.

Zugegebenermaßen ist die Abzählbarkeit ein sehr grobes Handwerkszeug. Denn wir unterscheiden nur zwei Stufen von Unendlichkeit. Aber für unsere Zwecke reicht das völlig aus.

3.3.1 Definition

Eine Menge A heißt *abzählbar (unendlich)*, wenn es eine bijektive Abbildung $f : A \to \mathbb{N}$ gibt. Wir schreiben dann: $|A| = |\mathbb{N}| = \omega$. Für eine endliche Menge A schreiben wir $|A| < \omega$. Für eine unendliche Menge A, die nicht abzählbar ist, schreiben wir $|A| > \omega$ und nennen A *überabzählbar*.

Beispiel

1. $|\{1, 2, 3, 4, 5\}| < \omega$
2. $|\mathbb{N}| = |\mathbb{Z}| = |\mathbb{Q}| = \omega$
3. Für ein endliches Alphabet Σ gilt: $|\Sigma^*| = \omega$
4. $|\mathbb{R}| > \omega$
5. $|\{g : \mathbb{N} \to \{0, 1\}\}| > \omega$

3.3.2 Beweistechniken für die Abzählbarkeit

Hier stellen wir kurz einige Techniken zum Nachweis der Abzählbarkeit einer Menge vor:

1. Explizite Angabe einer „Abzählungsfunktion“.
2. *Teilmengenbeziehung*: Für eine abzählbare Menge A ist jede Teilmenge $B \subseteq A$ abzählbar oder endlich.
3. Standardkonstruktionen (Beweis per Induktion usw.)

Beispielsweise kann man den Beweis für $|\mathbb{Z}| = |\mathbb{Q}| = \omega$ durch die explizite Angabe einer Abzählung führen. Um zu beweisen, daß die ganzen Zahlen $\mathbb{Z}$ abzählbar unendlich sind, definieren wir eine Funktion, die die ganzen Zahlen bijektiv auf die natürlichen Zahlen abbildet. In Abb. 1.2 ist diese Funktion skizziert.

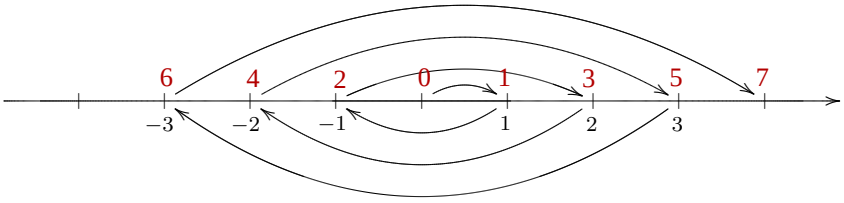


Abbildung 1.2. Die Skizze zeigt, dass $\mathbb{Z}$ abzählbar ist.

Für die rationalen Zahlen $\mathbb{Q}$ gehen wir ähnlich vor. Wir beweisen zunächst, daß die Menge aller Brüche abzählbar ist. Dazu tragen wir alle Brüche in ein Koordinatensystem ein, in dem die x-Achse den Zähler und die y-Achse den Nenner repräsentieren. In Abbildung 1.3 ist die entsprechende Abzählung skizziert. Da die Menge der rationalen Zahlen eine Teilmenge der Brüche ist, folgt mit der Teilmengenbeziehung, dass auch die Menge der rationalen Zahlen abzählbar ist.

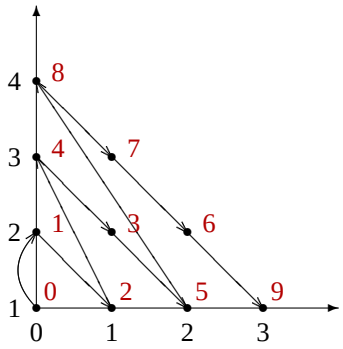


Abbildung 1.3. Die Skizze zeigt, dass die Menge der Brüche abzählbar ist.

3.3.3 Beweistechniken für die Überabzählbarkeit

Nun geben wir einige Beweistechniken für die Überabzählbarkeit einer Menge an.

1. *Obermengenbeziehung*: Für eine überabzählbare Menge A ist jede Obermenge $B \supseteq A$ überabzählbar.

   Beispiel: $|\mathbb{C}| > \omega$, da $|\mathbb{R}| > \omega$ und $\mathbb{C} \supseteq \mathbb{R}$

2. *Diagonalisierung*: Die Diagonalisierung läßt sich am besten an einem Beispiel erklären. Wir verwenden daher dieses Verfahren an dieser Stelle dazu, um die Aussage $|\{g : \mathbb{N} \to \{0,1\}\}| > \omega$ (vgl. Beispiel Punkt 5) zu beweisen.

   Das Diagonalisierungsverfahren ist ein Beweisverfahren durch Widerspruch. Wir nehmen also zunächst an, dass $A = \{g : \mathbb{N} \to \{0,1\}\}$ abzählbar ist. Wir können dann die Funktionen aus A abzählen: g_0, g_1, g_2, ... Alle Funktionen aus A sind also in der folgende (unendlichen) Tabelle repräsentiert:

	0	1	2	3	4	5	...
g_0	1	0	1	1	0	0	...
g_1	0	1	0	0	1	1	...
g_2	0	1	1	0	0	1	...
g_3	1	0	1	1	1	1	...
g_4	1	1	0	0	0	0	...
g_5	0	1	1	1	0	0	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

   Wir konstruieren nun eine Funktion $f : \mathbb{N} \to \{0,1\}$ entlang der Diagonalen (rot) wie folgt:

$$f(n) = \left\{ \begin{array}{ll} 1 & \text{falls } g_n(n) = 0 \\ 0 & \text{falls } g_n(n) = 1 \end{array} \right\} = \overline{g_n(n)}$$

   Offensichtlich ist f ein Element von A, kommt aber in der Abzählung g_0, g_1, g_2, ... nicht vor. Denn gäbe es ein g_i mit $g_i = f$, so müßte gelten: $g_i(i) = f(i) = \overline{g_i(i)} \neq g_i(i)$; dies ist offensichtlich ein Widerspruch.

Beobachtung: : Da Σ^* für jedes Alphabet abzählbar ist, können wir nur abzählbar viele Dinge (Automaten, Funktionen, Sprachen, ...) syntaktisch durch Zeichenreihen repräsentieren. Allerdings gibt es überabzählbar viele Funktionen und Sprachen. Damit existieren überabzählbar viele Funktionen, die wir nicht syntaktisch repräsentieren, geschweige denn berechnen können.

3.4 Grammatiken

Grammatiken gehören streng genommen nicht zu den Grundbegriffen, die wir voraussetzen. Sie sind zentraler Gegenstand der Vorlesung.
Da wir im Laufe der Vorlesung verschiedene Varianten von Grammatiken einführen, ist es zweckmäßig, hier die allgemeinste Form der Grammatik einzuführen. In den verschiedenen Kapiteln werden wir dann die verschiedenen Varianten als Spezialfall definieren. In der vollen Allgemeinheit werden wir die Grammatiken dann erst wieder gegen Ende der Vorlesung betrachten.

Motivation: Syntaktische Repräsentation einer Sprache (zum Beispiel: Programmiersprachen, syntaktische Ausdrücke, XML, ...). Ein typisches Beispiel ist die Backus-Naur-Form BNF:

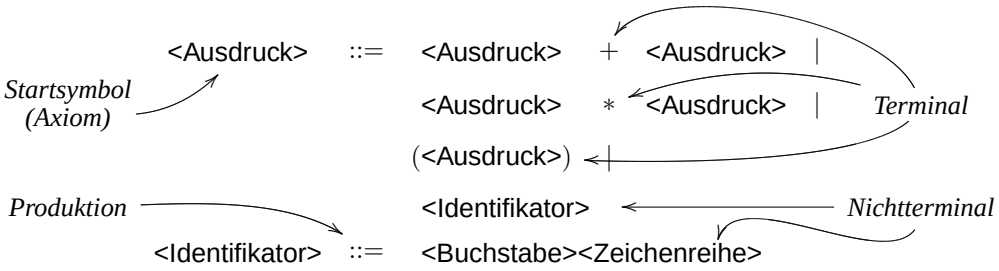


Abbildung 1.4. Eine Grammatik in Backus-Naur-Form.

Definition 1.1 *Eine* Grammatik G *über einem Alphabet* Σ *besteht aus*

- *einer endliche Menge* V *von* Variablen *(*Nonterminalen*) mit* $V \cap \Sigma = \varnothing$*,*

  Ein Elemente von Σ *nennen wir auch* Terminalsymbol *und* Σ *das* Terminalalphabet.

- *einer endlichen binären Relation* $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$*, den* Produktionen *oder* Regeln *und*

- *einer* Startvariablen *(einem* Startsymbol, Axiom*)* $S \in V$*.*

Wir notieren die Grammatik durch $G = (V, \Sigma, P, S)$*. Die Regeln* $(\alpha, \beta) \in P$ *einer Grammatik notieren wir meist durch* $\alpha \to \beta$*.*
Die Elemente von Σ *nennen wir auch* Terminale*.*

Bemerkung: „Eine Grammatik ist kein 4-Tupel!“
Der 4-Tupel ist lediglich eine mathematische Formalisierung. Eine Grammatik ist ein Konzept zur Beschreibung von (formalen) Sprachen) durch Regeln.

Beispiel 1.1 (Grammatik)
Als Beispiel für die mathematische Formulierung einer Grammatik geben wir die Grammatik zur BNF in Abb. 1.4 an:

$$\begin{array}{lcl} \Sigma & = & (,),+,*,a,b,c\} \\ V & = & \{A,I\} \\ P & = & \{A \to A+A, A \to A*A, A \to (A), A \to I, I \to a, I \to b, I \to c\} \\ G & = & (V,\Sigma,P,A) \end{array}$$

Frage: Welche Sprache wird durch G definiert?

Antwort: Definition über *Ableitung* von *Satzformen* ausgehend vom Axiom:

$$\begin{array}{llllll} A & \Rightarrow_G & A+A & \Rightarrow_G & A+(A) & \Rightarrow_G \quad I+(A) \\ & \Rightarrow_G & a+(A) & \Rightarrow_G & \underbrace{a+(A+A)}_{\textit{Satzform von } G} & \Rightarrow_G^* \quad \underbrace{a+(a+a)}_{\textit{Satz von } G} \end{array}$$

Definition 1.2 (Ableitung, Satzform, Satz, Sprache)
Sei $G = (V, \Sigma, P, S)$ eine Grammatik und seien $\alpha, \beta \in (V \cup \Sigma)^*$:

1. *β heißt aus α* direkt ableitbar*, wenn zwei Wörter $\gamma, \gamma' \in (V \cup \Sigma)^*$ und eine Produktion $\alpha' \to \beta' \in P$ existieren, so daß $\alpha = \gamma\alpha'\gamma'$ und $\beta = \gamma\beta'\gamma'$ gilt.*

   Wir schreiben dann $\alpha \Rightarrow_G \beta$. Den Index G lassen wir auch weg, wenn er aus dem Kontext hervorgeht.

   β heißt aus α ableitbar*, wenn $\alpha \Rightarrow_G^* \beta$ gilt. Dabei ist $\Rightarrow_G^*$ die transitiv-reflexive Hülle von $\Rightarrow_G$.*

2. *α heißt* Satzform *von G, wenn $S \Rightarrow_G^* \alpha$ gilt.*

3. *w heißt* Satz *von G, wenn w eine Satzform von G ist und wenn gilt: $w \in \Sigma^*$.*

   Die Menge aller Sätze von G heißt die von G erzeugte Sprache *und wird mit $L(G)$ bezeichnet (d.h. $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$).*

Bemerkung: Die Ableitung eines Satzes kann auch als Graph dargestellt werden. Für kontextfreie Grammatiken (d.h. für Grammatiken mit $P \subseteq V \times (V \cup \Sigma)^*$) läßt sich die Ableitung sogar als Baum darstellen (siehe Abschnitt 1.2).

3.5 Konventionen

Im folgenden werden wir (soweit wie möglich) die Bezeichner in einer einheitlichen Weise benutzten. Dies erlaubt es uns, aus dem Bezeichner auf den „Typ“ der bezeichneten Objekte zu schließen. Wir werden dann nach einer gewissen Eingewöhnungszeit auf die explizite Angabe ihres Typs verzichten.

- Σ, Δ sowie Σ_i, Σ' etc. bezeichnen ein Alphabet.
- $a, b, c, \ldots$ sowie a_i, a' etc. bezeichnen ein Zeichen eines Alphabetes.
- u, v, w, x, y, z sowie w_1, w' etc. bezeichnen ein Wort über einem Alphabet.
- α, β, γ sowie α_i, α' etc. bezeichnen ein Wort über einem verallgemeinerten Alphabet (z.B. die Satzformen einer Grammatik).
- $L, L_i, L', \ldots$ bezeichnen eine Sprach über einem Alphabet.
- A, B, C, X, Y, Z sowie A_i, A' etc. bezeichnen eine Variable einer Grammatik.
- S sowie S_i, S' etc. bezeichnen die Startvariable einer Grammatik.
- G sowie G_i, G' etc. bezeichnen eine Grammatik.
- V sowie V_i, V' etc. bezeichnet die Variablenmenge einer Grammatik.
- P sowie P_i, P' etc. bezeichnen die Produktionsmenge (Regeln) einer Grammatik.

Diese Konventionen werden wir später erweitern, wenn wir später weitere Konzepte einführen. Leider lassen sich dann Mehrdeutigkeiten nicht ganz vermeiden.

Teil B

Automaten und Formale Sprachen

Kapitel 2

Reguläre Sprachen

1 Endliche Automaten

1.1 Motivation

Beispiel 2.1
Charakterisierung der gültigen Bezeichner einer Programmiersprache durch endliche Automaten:

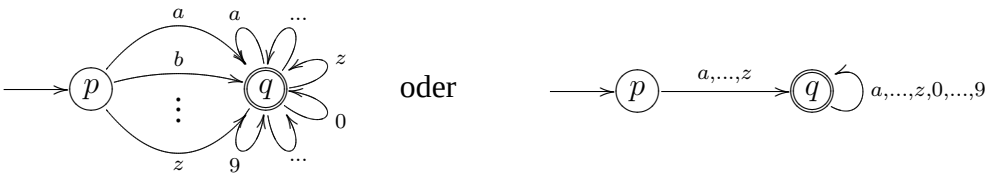


Abbildung 2.1. Zwei verschiedene Darstellungen eines Automaten für das Erkennen von Bezeichnern.

Beispiel 2.2
Charakterisierung aller Wörter über $\{0, 1\}$, die mit mindestens drei Einsern endet. Diese Menge wird durch den folgenden Automaten beschrieben bzw. von ihm akzeptiert:

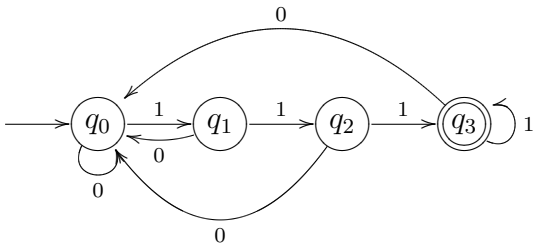


Abbildung 2.2. Ein deterministischer Automat für Wörter, die mit mindestens drei Einsern enden.

Für manche Sprachen ist der entsprechende deterministische Automat sehr groß. Oft sind nichtdeterministischen Automaten kleiner, übersichtlicher und einfacher zu konstruieren. Der nichtdeterministische Automat für die obige Sprache ist in Abbildung 2.3 dargestellt.

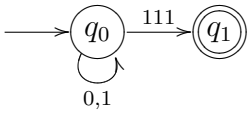


Abbildung 2.3. Ein nicht-deterministischer Automat für Wörter, die mit mindestens drei Einsern enden.

Bemerkungen:

- Ein nicht-deterministischer Automat ist oft einfacher und übersichtlicher als der entsprechende deterministische Automat.
- Ein nicht-deterministischer Automat ist oft einfacher zu konstruieren (aus anderen Repräsentationen bzw. aus einer informellen Beschreibung der Sprache) als ein entsprechender deterministischer Automat.
- Zum Erkennen einer Sprache ist ein deterministischer Automat besser geeignet.
- Manche Operationen auf Sprachen lassen sich einfacher mit deterministischen Automaten durchführen (beispielsweise die Konstruktion eines Automaten für das Komplement einer Sprache); andere Operationen auf Sprachen lassen sich einfacher mit nicht-deterministischen Automaten durchführen (beispielsweise die Konstruktionen eines Automaten, der die Vereinigung zweier Sprachen akzeptiert).

Deshalb sind beide Varianten sinnvoll. Besonders schön ist es, wenn man für jeden nicht-deterministischen Automaten *effektiv*[1] einen entsprechenden deterministischen Automaten konstruieren kann. Wir werden sehen, daß diese Überführung für endliche Automaten immer effektiv möglich ist. Für andere Automatentypen ist dies nicht immer möglich; beispielsweise kann nicht jeder nicht-deterministische Kellerautomat in einen entsprechenden deterministischen Kellerautomaten überführt werden (vgl. Kapitel 3 Abschnitt 2)

1.2 Verschiedene Definitionen endlicher Automaten

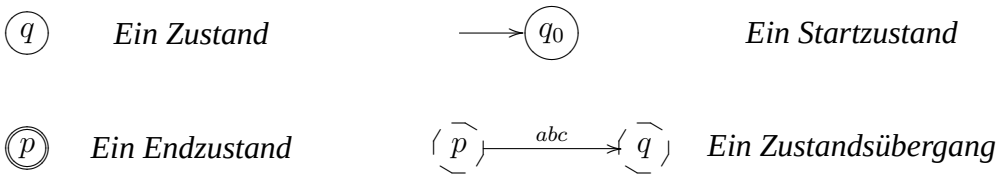
Wir beginnen mit der allgemeinsten Variante der endlichen Automaten und schränken diese dann immer weiter ein. Aller weiteren Varianten sind also Spezialfälle.

Definition 2.1 (Endlicher Automat (EA))
Ein endlicher Automat *(EA)* A *über einem Alphabet* Σ *besteht aus*

- *einer endlichen Menge* Q *von* Zuständen,
- *einer Menge von* Startzuständen I *mit* $I \subseteq Q$,
- *einer endlichen Menge von* Zustandsübergängen $\delta \subseteq Q \times \Sigma^* \times Q$ *und*
- *einer Menge von* Endzuständen F *mit* $F \subseteq Q$.

[1] Effektiv bedeutet dabei, daß es ein automatisches Verfahren zur Überführung eines nicht-deterministischen Automaten in einen deterministischen gibt. Noch schöner ist es, wenn das Verfahren effizient durchgeführt werden kann (Achtung: effektiv $\neq$ effizient).

Wir schreiben $A = (Q, \Sigma, \delta, I, F)$. Die einzelnen Elemente eines Automaten stellen wir graphisch wie folgt dar:



Bemerkung:

- Oft wird δ als Abbildung formalisiert. Dies ist schon bei nicht-deterministischen Automaten unschön; bei unserer noch etwas allgemeineren Variante (wir erlauben Wörter an den Kanten) sehr unzweckmäßig. Deshalb formalisieren wir δ als Relation (das Konzept ist aber in beiden Fällen dasselbe).

- Bei uns sind alle weiteren Varianten Spezialfälle der obigen Definition. Deshalb haben wir auf das Attribut „nicht-deterministisch“ zunächst verzichtet. In der Literatur wird unsere Variante aber oft als *nicht-deterministischer endlicher Automat* (NEA) eingeführt.

- Ein Wort w wird von einem Automaten *erkannt* (gehört zur akzeptierten Sprache des Automaten), wenn es im Automaten einen Pfad $q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} q_2 \dots q_{n-1} \xrightarrow{w_n} q_n$ von einem Startzustand $q_0 \in I$ zu einem Endzustand $q_n \in F$ mit $w = w_1 w_2 \dots w_n$ gibt.

  · *Der gleiche Zustand kann auf dem Pfad mehrmals vorkommen.*
  · *Der Pfad kann die Länge 0 haben (d.h. $q_0 = q_n \in I \cap F$); dann gilt $w = \varepsilon$.*

Für eine Zustandsübergangsrelation $\delta \subseteq Q \times \Sigma^* \times Q$ definieren wir die Relation $\delta^* \subseteq Q \times \Sigma^* \times Q$ als die kleinste Relation mit:

- $\delta \subseteq \delta^*$
- für alle $q \in Q$ gilt $(q, \varepsilon, q) \in \delta^*$
- für alle $p, q, r \in Q$ und $u, v \in \Sigma^*$ mit $(p, u, q) \in \delta^*$ und $(q, v, r) \in \delta^*$ gilt auch $(p, uv, r) \in \delta^*$.

Alternativ dazu können wir δ^* über die Hilfsrelationen δ^i. Die δ^i sind induktiv wie folgt definiert:

- $\delta^0 = \{(q, \varepsilon, q) \mid q \in Q\}$,
- $\delta^1 = \delta$,
- $\delta^{i+1} = \{(p, uv, r) \mid (p, u, q) \in \delta, (q, v, r) \in \delta^i\}$

Damit läßt sich dann δ^* wie folgt definieren $\delta^* = \bigcup_{i \in \mathbb{N}} \delta^i$.

In Abbildung 2.4 ist die Idee dieser Definition nochmals graphisch dargestellt.

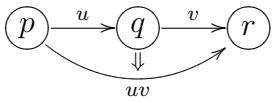


Abbildung 2.4. Graphische Darstellung der Definition von δ^*.

Definition 2.2 (Akzeptierte Sprache) *Sei $A = (Q, \Sigma, \delta, I, F)$ ein endlicher Automat. Die von A* akzeptierte Sprache $L(A)$ *ist definiert durch:*

$$L(A) = \{w \in \Sigma^* \mid \exists p \in I, q \in F : (p, w, q) \in \delta^*\}$$

$L(A)$ heißt auch die Leistung *von A. Für $w \in L(A)$ sagen wir auch, daß w von A* akzeptiert *oder* erkannt *wird.*

Beispiel 2.3
Die akzeptierte Sprache der beiden endlichen Automaten aus Beispiel 2.2 ist: $\{w111 \mid w \in \Sigma^*\}$

Definition 2.3 (Spezialfälle endlicher Automaten)
Ein endlicher Automat $A = (Q, \Sigma, \delta, I, F)$ heißt:

1. *Automat* mit (genau) einem Startzustand*, wenn gilt $|I| = 1$; er heißt*
2. alphabetisch*, wenn gilt $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$; er heißt*
3. buchstabierend*, wenn gilt $\delta \subseteq Q \times \Sigma \times Q$; er heißt*
4. deterministisch *(DEA), wenn er buchstabierend ist, genau einen Startzustand besitzt und für alle $p, q, r \in Q$ und alle $a \in \Sigma$ mit $(p, a, q) \in \delta$ und $(p, a, r) \in \delta$ gilt $q = r$; der Automat heißt*
5. vollständig*, wenn für jedes $p \in Q$ und jedes $a \in \Sigma$ ein $q \in Q$ mit $(p, a, q) \in \delta$ existiert.*

Bemerkungen: Die Spezialfälle der Automaten schränken im wesentlich die Zustandsübergangsrelation δ ein. Einige Eigenschaften übertragen sich auf δ^* bzw. implizieren Eigenschaften von δ^*:

1. Wenn A alphabetisch ist, dann gilt: für $(p, w, r) \in \delta^*$ und jede Zerlegung u, v von w (d.h. $uv = w$) gibt es ein $q \in Q$ mit $(p, u, q) \in \delta^*$ und ein $(q, v, r) \in \delta^*$
2. Wenn A deterministisch ist, dann gilt für alle $(p, w, q) \in \delta^*$ und alle $(p, w, r) \in \delta^*$ auch $q = r$.
3. Wenn A vollständig und deterministisch ist, dann existiert für jedes $q \in Q$ und jedes $w \in \Sigma$ genau ein $r \in Q$ mit $(q, w, r) \in \delta^*$.

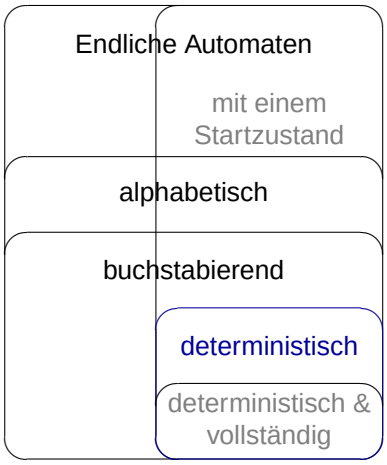


Abbildung 2.5. Syntaktische Inklusionsbeziehungen zwischen den Spezialfällen.

Bemerkungen:

1. Abbildung 2.5 gibt eine Übersicht über die verschiedenen Spezialfälle endlicher Automaten. Die Abbildung zeigt, die Inklusionsbeziehungen, die sich aufgrund der syntaktischen Einschränkungen ergeben.

2. Für einen Automaten mit (genau) einem Startzustand $q_0 \in Q$ schreiben wir auch $A = (Q, \Sigma, \delta, q_0, F)$ anstelle von $A = (Q, \Sigma, \delta, \{q_0\}, F)$.

3. Endliche Automaten ohne Einschränkungen oder mit den Einschränkungen 1 - 3 aus Definition 2.3 werden auch *nicht-deterministische endliche Automaten* (NEA) genannt.

   Achtung: Hier gibt es einen kleinen sprachlichen Stolperstein: Die Menge aller Automaten nennen wir auch nicht-deterministische Automaten. Wenn wir das Komplement der Menge der deterministischen Automaten meinen, nennen wir sie nicht deterministische Automaten. Nicht-deterministische Automaten können dagegen deterministisch sein!
   Den Bindestrich übersieht man leicht; hören kann man ihn ohnehin nicht. In der Praxis ist dies aber selten ein Problem, da es sich meist aus dem Kontext ergibt, ob wir über die nicht-deterministischen Automaten (also die Menge aller endlichen Automaten) oder die nicht deterministischen (also die Menge aller endlichen Automaten ohne die deterministischen Automaten) reden.

4. Alphabetische Automaten können, im Gegensatz zu den buchstabierenden Automaten, sogenannte ε-*Übergänge* enthalten: $p \xrightarrow{\varepsilon} q$.

5. Bei deterministischen Automaten wird δ meist als partielle Abbildung $\delta : Q \times \Sigma \to Q$ aufgefaßt; bei vollständigen deterministischen Automaten ist $\delta : Q \times \Sigma \to Q$ eine totale Abbildung. Wir sprechen dann auch von einer partiellen bzw. totalen *Übergangsfunktion*.

   In manchen Lehrbüchern wird die Vollständigkeit bereits in der Definition des deterministischen Automaten verlangt. Da die Vollständigkeit und der Determinismus zwei unabhängige Konzepte sind, haben wir sie separat formalisiert.

Konventionen Hier ergänzen wir unsere Konventionen zur Benutzung von Bezeichnern.

- A, A_i, A' etc. bezeichnen einen Automaten [2]
- p, q, r sowie q_i, q' etc. bezeichnen den Zustand eines Automaten.

  q_0 bezeichnet den Startzustand eines Automaten (mit genau einem Startzustand.
- $\delta, \delta_i, \delta'$ bezeichnen die Zustandsübergangsrelation eines Automaten.
- I bezeichnet die Menge der Startzustände eines Automaten.
- F bezeichnet die Menge der Endzustände eines Automaten.

1.3 Äquivalenz der verschiedenen Automatentypen

Wir werden sehen, daß sich die Sprachen, die sich mit den verschiedenen Spezialfällen von Automaten erkennen lassen, nicht unterscheiden. Semantisch sind die verschiedenen Spezialfälle von Automaten also alle äquivalent zueinander.

Definition 2.4 (Äquivalenz endlicher Automaten)
Zwei Automaten heißen äquivalent*, wenn sie dieselbe Sprache akzeptieren.*

Satz 2.5 *Zu jedem endlichen Automaten gibt es effektiv einen äquivalenten endlichen Automaten, der deterministisch und vollständig ist.*

> *Die effektive Existenz bedeutet, daß man den äquivalenten deterministischen und vollständigen endlichen Automaten automatisch konstruieren kann. Die Konstruktion werden wir im Beweis kennenlernen.*

Beweis: Der Beweis ist relativ lang und enthält verschiedene Konstruktionen, die auch einzeln von Interesse sind. Deshalb geben wir zunächst einen Überblick über die verschiedenen Konstruktionsschritte, die dann im Detail ausgeführt werden:

1. Schritt: Endlicher Automat $\rightsquigarrow$ Automat mit genau einem Startzustand.

2. Schritt: Automat mit einem Startzustand $\rightsquigarrow$ alphabetischer Automat mit einem Startzustand.

3. Schritt (ε-Elimination): Alphabetischer Automat mit einem Startzustand $\rightsquigarrow$ buchstabierender Automat mit einem Startzustand.

4. Schritt (Potenzautomatenkonstruktion): Buchstabierender Automaten mit einem Startzustand $\rightsquigarrow$ vollständiger und deterministischer endlicher Automat.

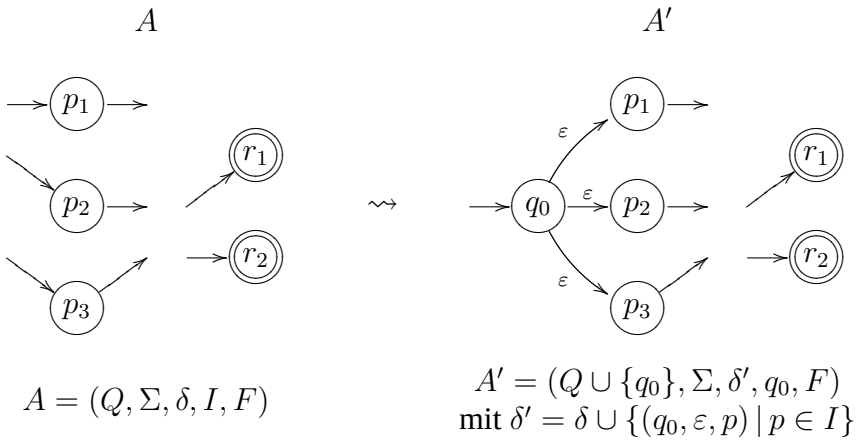


Abbildung 2.6. Schritt 1: Konstruktion eines äquivalenten Automaten mit genau einem Startzustand.

Schritt 1: Erzeuge aus einem endlichen Automaten (mit mehreren Startzuständen) einen endlichen Automaten mit genau einem Startzustand.
Idee: Füge einen neuen Startzustand q_0 zum Automaten hinzu, von dem jeweils einen ε-Übergang von q_0 zu jedem bisherigen Startzustand führt. Wähle q_0 als einzigen Startzustand.
Diese Konstruktion ist in Abb. 2.6 graphisch dargestellt und mathematisch formalisiert. Offensichtlich gilt $L(A) = L(A')$.

Entsprechend könnten wir einen äquivalenten Automaten mit genau einem Startzustand und genau einem Endzustand konstruieren. Da die noch spezielleren Automaten jedoch im allgemeinen mehr als einen Endzustand besitzen müssen, verzichten wir darauf, diese Konstruktion zu formalisieren. Denn bei den weiteren Konstruktionen werden wir wieder zusätzliche Endzustände einführen (z.B. in Schritt 3).

Schritt 2: Erzeuge aus einem endlichen Automaten mit genau einem Startzustand einen alphabetischen endlichen Automaten mit genau einem Startzustand.
Idee: Teile einen Übergang mit einem Wort $w = a_1 a_2 \dots a_n$ mit $n > 1$ in mehrere Übergänge mit jeweils nur einem Zeichen auf. Dazu müssen neue Zwischenzustände eingeführt werden. Die Konstruktion ist in Abb. 2.7 graphisch dargestellt.

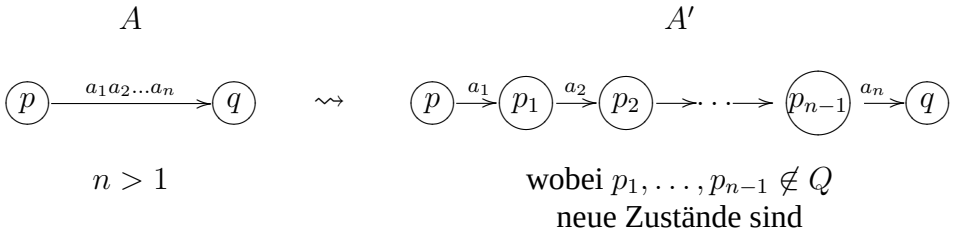


Abbildung 2.7. Konstruktion eines äquivalenten alphabetischen Automaten.

Schritt 3 (ε-Elimination): Erzeuge aus einem alphabetischen endlichen Automaten mit genau einem Startzustand einen buchstabierenden endlichen Automaten mit genau einem Startzustand.

[2] Zunächst verstehen wir darunter nur endliche Automaten; wir werden den Bezeichner A jedoch später auch für andere Automaten benutzten.

Idee: Wir löschen die ε-Übergänge. Wir müssen jedoch sicherstellen, daß sich dabei die akzeptierte Sprache des Automaten nicht ändert. Deshalb fügen wir im Schritt a. neue Kanten hinzu und im Schritt b. zeichnen wir einige neue Endzustände aus. Erst im Schritt c. löschen wir dann die ε-Übergänge. Diese drei Schritte sind in Abb. 2.8 graphisch dargestellt:

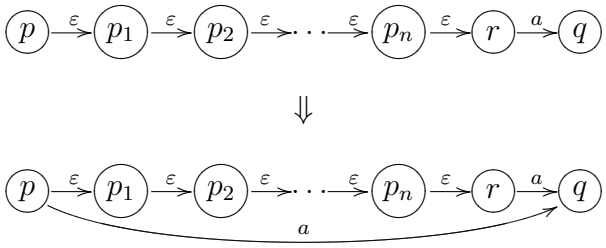
Schritt a. Für jede Kette von ε-Übergängen von p nach r gefolgt von einem a-Übergang nach q fügen wir einen a-Übergang direkt von p nach q hinzu.

Schritt b. Falls von p eine Kette von ε-Übergängen zu einem Endzustand q führt, wird auch p ein Endzustand.

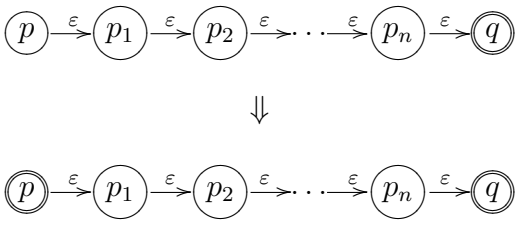
Schritt c. Löschen der ε-Übergänge.

Die Schritte a. und b. können hierbei beliebig gemischt werden. Schritt c., d.h. das Entfernen der ε-Übergänge, darf jedoch erst durchgeführt werden, wenn durch Schritt a. und b. keine neuen Kanten bzw. Endzustände mehr hinzugefügt werden können.

a. Einfügen eines a-Übergangs für eine $\varepsilon \ldots \varepsilon a$-Kette:



b. Auszeichnung eines weiteren Endzustandes, wenn eine ε-Kette zu einem Endzustand existiert:



c. Löschen der ε-Übergänge:

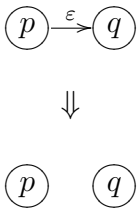


Abbildung 2.8. Konstruktion eines buchstabierenden Automaten aus einem alphabetischen Automaten.

Durch diese Transformation wird aus einem Automaten $A = (Q, \Sigma, \delta, I, F)$ ein Automat $A' = (Q, \Sigma, \delta', I, F')$ erzeugt mit:

$$\delta' = (\delta \cup \underbrace{\{(p, a, q) \mid (p, \varepsilon, r) \in \delta^*, (r, a, q) \in \delta\}}_{\text{Schritt a.}}) \setminus \underbrace{(Q \times \{\varepsilon\} \times Q)}_{\text{Schritt c.}}$$

$$F' = F \cup \underbrace{\{p \in Q \mid (p, \varepsilon, q) \in \delta^*, q \in F\}}_{\text{Schritt b.}}$$

Schritt 4 (Potenzautomatenkonstruktion): Erzeuge aus einem endlichen nicht-deterministischen Automaten mit einem Startzustand einen endlichen deterministischen Automaten.
Idee (*Potenzautomat*): Wir verfolgen alle Wege im endlichen buchstabierenden Automaten gleichzeitig, dabei merken wir uns alle Zustände, in denen sich der Automat befinden könnte.

Beispiel
Als Beispiel dazu betrachten wir den buchstabierenden Automaten aus Abb. 2.9, der die Sprache aller Wörter akzeptiert, die mit mind. drei Einsern enden.

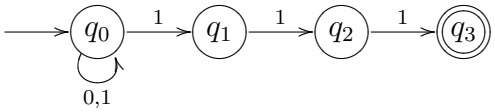


Abbildung 2.9. Ein buchstabierender Automat für Wörter, die mit mindestens drei Einsern enden.

Nun betrachten wir das Wort $w = 110111$ und simulieren den Automaten:

initial : $\{q_0\}$	
$1 : \{q_0, q_1\}$	Übergang von q_0 mit 1 nach q_0 oder q_1 möglich.
$1 : \{q_0, q_1, q_2\}$	Übergang von q_0 mit 1 nach q_0 oder q_1 möglich. Übergang von q_1 mit 1 nach q_2 möglich.
$0 : \{q_0\}$	Einen Übergang mit 0 gibt es nur von q_0; der führt wieder nach q_0.
$1 : \{q_0, q_1\}$	...
$1 : \{q_0, q_1, q_2\}$	...
$1 : \{q_0, q_1, q_2, \underline{\underline{q_3}}\}$	Automat kann sich (bei geeigneter Auswahl der Alternativen) im Endzustand q_3 befinden; das Wort w wird also akzeptiert.

Die Simulation aller möglichen erreichbaren Zustände für jedes beliebige Eingabewort w können wir nun ein für alle mal durchführen: Abb. 2.10 zeigt den entsprechenden Automaten.

Formal läßt sich die Potenzautomatenkonstruktion wie folgt beschreiben: Für einen buchstabierenden Automaten $A = (Q, \Sigma, \delta, q_0, F)$ definieren wir $A' = (2^Q, \Sigma, \delta', \{a_0\}, F')$, wobei $2^Q = \{P \subseteq Q\}$ die Potenzmenge von Q ist, d.h. die Menge aller Teilmengen von Q.
Die Potenzmenge 2^Q von Q ist zwar sehr groß; sie ist aber endlich. Also ist A' auch ein endlicher Automat.

Achtung: Die Bezeichner P und R bezeichnen im folgenden Teilmengen von Q, d.h. Zustände des Potenzautomaten.

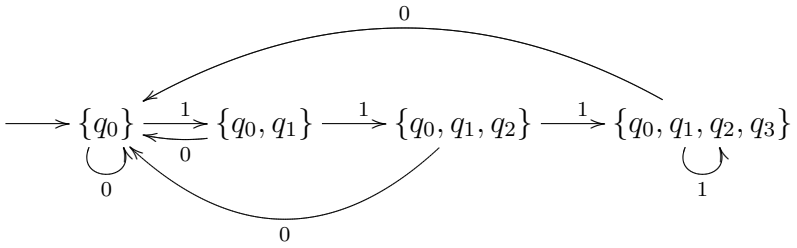


Abbildung 2.10. Ein deterministischer Automat für Wörter, die mit mindestens drei Einsern enden.

Die Übergangsrelation δ' wird wie folgt definiert: $(P, a, R) \in \delta'$ gdw. $R = \{r \in Q \mid \exists p \in P : (p, a, r) \in \delta)\}$; die Menge R ist also die Menge aller Zustände r, die von irgendeinem Zustand $p \in P$ aus mit einem a-Übergang erreichbar sind.
Die Menge der Endzustände von A' ist definiert durch: $F' = \{P \subseteq Q \mid P \cap F \neq \varnothing\}$; d.h. diejenigen Teilmengen von Q, die mind. einen Endzustand enthalten.
Offensichtlich ist A' deterministisch (für P und a ist R eindeutig definiert) und vollständig.
Außerdem gilt $(\{q_0\}, w, P) \in \delta'^*$ gdw. $P = \{p \in Q \mid (q_0, w, p) \in \delta^*\}$ (Beweis durch Induktion). Damit gilt dann offensichtlich $L(A) = L(A')$.

Insgesamt haben wir also in vier Schritten aus einem beliebigen endlichen Automaten effektiv einen äquivalenten deterministischen und vollständigen Automaten konstruiert.

Achtung: Wir haben nirgends explizit dafür gesorgt, daß der resultierende Automat vollständig ist. Dies ist aber ein Nebeneffekt der Potenzautomatenkonstruktion. Denn für jede Teilmenge $P \subseteq Q$ und jedes Zeichen $a \in \Sigma$ gibt es ein $R \subseteq Q$ mit $(P, a, R) \in \delta'$. Wenn es von keinem $p \in P$ einen a-Übergang gibt, dann gilt $R = \varnothing$ (vgl. Definition von δ').

□

Bemerkungen:

- Im allgemeinen hat der Potenzautomat (wie wir ihn formal definiert haben) auch Zustände, die nicht vom Startzustand aus erreichbar sind (das können sehr viele sein). Deshalb wird der Potenzautomat in der Praxis ausgehend vom Startzustand konstruiert und es werden nur solche Zustände erzeugt, die vom Startzustand aus erreichbar sind.

- Wenn eine Sprache von einem endlichen Automaten akzeptiert wird, dann können wir im folgenden „ohne Beschränkung der Allgemeinheit“ (o.B.d.A) davon ausgehen, dass sie von einem deterministischen und vollständigen endlichen Automaten akzeptiert wird.

1.4 Einfache Abschlußeigenschaften

Frage: Welche Operationen auf von endlichen Automaten erkannten Sprachen liefern wieder eine von einem endlichen Automaten erkannte Sprache ?

Satz 2.6 (Abschluß unter $\cap$, $\cup$, $\neg$, $\circ$ und *) *Seien L_1 und L_2 zwei Sprachen, die von einem endlichen Automaten akzeptiert werden. Dann werden die Sprachen*

1. $L_1 \cup L_2$,
2. $L_1 \cap L_2$,
3. $\overline{L_1}$ *(d.h.* $\overline{L_1} = \Sigma^* \setminus L_1$*)*
4. $L_1 \circ L_2$ *und*
5. L_1^*

ebenfalls von einem endlichen Automaten akzeptiert.

Die Existenz dieser Automaten ist effektiv! D.h. wir können aus den Automaten für L_1 und L_2 jeweils auch einen Automaten konstruiert, der die oben genannten Sprachen akzeptiert.

Beweis: Hier werden wir nur 1. und 4. beweisen, die anderen Beweise werden wir in der Übung führen. Für die Beweise von 2., 3. und 5. geben wir nach dem Beweis ein paar Hinweise.

$L_1 \cup L_2$:
Sei $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$ ein endlicher Automat mit $L_1 = L(A_1)$ und $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$ ein endlicher Automat mit $L_2 = L(A_2)$. Ohne Beschränkung der Allgemeinheit gilt $Q_1 \cap Q_2 = \varnothing$. Wir definieren:

$$A = (\underbrace{Q_1 \cup Q_2}_{Q}, \Sigma, \underbrace{\delta_1 \cup \delta_2}_{\delta}, \underbrace{I_1 \cup I_2}_{I}, \underbrace{F_1 \cup F_2}_{F})$$

Offensichtlich gibt es für jedes $w \in \Sigma^*$ genau dann ein $p \in I$ und ein $q \in F$ mit $(p, w, q) \in \delta^*$, wenn $p_1 \in I_1$ und ein $q_1 \in F_1$ mit $(p_1, w, q_1) \in \delta_1^*$ existieren oder wenn $p_2 \in I_2$ und $q_2 \in F_2$ mit $(p_2, w, q_2) \in \delta_2^*$ existieren. Also gilt $L(A) = L(A_1) \cup L(A_2)$.

$L_1 \circ L_2$:
Seien A_1 und A_2 wie oben definiert. Wir definieren:

$$A = (\underbrace{Q_1 \cup Q_2}_{Q}, \Sigma, \underbrace{\delta_1 \cup (F_1 \times \{\varepsilon\} \times I_2) \cup \delta_2}_{\delta}, \underbrace{I_1}_{I}, \underbrace{F_2}_{F})$$

Dann gilt $L(A) = L(A_1) \circ L(A_2)$.

Redaktioneller Hinweis: Zu den Konstruktionen im Beweis sollten noch die entsprechenden Graphiken hinzugefügt werden.

□

Hinweise: Zur Konstruktion der Automaten für die anderen Operationen geben wir hier die wesentlichen Ideen an:

3. $\rightarrow$ deterministischer und vollständiger endlicher Automat.

2. $\rightarrow$ Verwendung von 1 und 3 (und de Morgan).

5. $\rightarrow$ Ähnlich wie 4., aber ε nicht vergessen!

Weitere Operationen auf Sprachen werden wir später noch untersuchen:

- *Homomorphismen*
- *inverse Homomorphismen*
- *Substitution*
- *Quotient*
- *Spiegelung*

2 Reguläre Ausdrücke

Motivation: In Abschnitt 1.4 haben wir bereits gesehen, dass die von endlichen Automaten akzeptierten Sprachen unter den sogenannten *regulären Operationen* abgeschlossen sind. In diesem Abschnitt werden wir den Zusammenhang zu der Sprachklasse, die sich über diese Operationen definieren lassen, den *regulären Sprachen*, herstellen. Es wird sich herausstellen, daß dies genau die Sprachklasse ist, die von endlichen Automaten akzeptiert wird.

2.1 Syntax und Semantik

Hier definieren wir die Syntax und die Semantik der regulären Ausdrücke. In dieser Definition halten wir Syntax und Semantik der Ausdrücke noch deutlich auseinander. Später, wenn wir uns dieses Unterschieds bewußt sind, werden wir diese Unterscheidung nicht mehr so klar vornehmen.

Definition 2.7 (Reguläre Ausdrücke und Sprachen)

Die Menge der regulären Ausdrücke *über einem Alphabet* Σ *ist induktiv definiert durch:*

1. $\varnothing$ *ist ein regulärer Ausdruck.*
2. ε *ist ein regulärer Ausdruck.*

   ε könnten wir uns auch sparen, da (semantisch) gilt $\{\varepsilon\} = \varnothing^*$.
3. *Für* $a \in \Sigma$ *ist* a *ein regulärer Ausdruck.*
4. *Für zwei reguläre Ausdrücke* r *und* s *sind*
   - $(r + s)$,
   - (rs) *und*
   - (r^*)

   reguläre Ausdrücke.

} *Syntax !*

Um die Lesbarkeit von regulären Ausdrücken zu erhöhen, geben wir der Operation * *eine höhere Priorität als* $\circ$*; der Operation* $\circ$ *geben wir eine höhere Priorität als* $+$*. Dadurch vermeiden wir unlesbare „Klammergebirge“. Beispielsweise steht dann* $a + bcd^*$ *für* $(a + (b(c(d^*))))$.

Die Semantik (Bedeutung) eines regulären Ausdrucks ist eine Sprache über Σ*. Die* durch einen regulären Ausdruck bezeichnete Sprache *ist induktiv definiert durch*

1. $\varnothing$ *bezeichnet die leere Sprache* $\varnothing$
2. ε *bezeichnet die Sprache* $\{\varepsilon\}$
3. a *bezeichnet die Sprache* $\{a\}$
4. *Bezeichnen* r *die Sprache* R *und* s *die Sprache* S*, dann bezeichnen*

   $\underbrace{\begin{matrix}(r+s)\\(rs)\\(r^*)\end{matrix}}_{\text{Syntax}}$ *die Sprache* $\underbrace{\begin{matrix}R \cup S\\R \circ S\\R^*\end{matrix}}_{\text{Semantik}}$

} *Semantik!*

Eine Sprache heißt regulär*, wenn sie durch einen regulären Ausdruck bezeichnet wird.*

2.2 Satz von Kleene

Nun kommen wir zum zentralen Satz über den Zusammenhang zwischen den regulären Sprachen und den von endlichen Automaten akzeptierten Sprachen: Sie sind gleich.

Das ist auch der Grund dafür, daß wir den „von endlichen Automaten akzeptierten Sprachen“ keinen eigenen Namen gegeben haben.

Satz 2.8 (Satz von Kleene)
Eine Sprache ist genau dann regulär, wenn sie von einem endlichen Automaten akzeptiert wird.

Beweis: Wir beweisen die beiden Richtungen der Genau-dann-wenn-Beziehung einzeln:
„⇒“: Wir beweisen induktiv über den Aufbau der regulären Ausdrücke, dass jede Sprache, die von einem regulären Ausdruck bezeichnet wird, auch von einem endlichen Automaten akzeptiert wird:

Zur Konstruktion der Automaten für die regulären Operatoren wenden wir den Satz 2.6 an:

- $(r + s)$ Gemäß Satz 2.6.1 können wir aus den Automaten für R und S, die gemäß Induktionsvoraussetzung existieren, den Automaten für $R \cup S$ konstruieren.
- (rs) Analog mit Satz 2.6.4.
- (r^*) Analog mit Satz 2.6.5.

„⇐“: Hier skizzieren wir nur die Idee: Wir konvertieren den Automaten in einen Automaten mit genau einem Startzustand q_0 und einem Endzustand q_n (vergleiche Anmerkung zu Schritt 1 im Beweis von Satz 2.5).
Diesen Automaten überführen wir nun schrittweise in einen *verallgemeinerten Automaten*, an dessen Kanten reguläre Ausdrücke stehen. Dabei eliminieren wir schrittweise jeden Zustand $q_1, \ldots, q_{n-1}$ (also jeden Zustand außer dem Start- und dem Endzustand) wie in Abbildung 2.11 dargestellt. Dabei wird sichergestellt, daß alle Wörter, die bei Übergängen zwischen den verbleibenden Zuständen ursprünglich möglich waren auch, weiterhin repräsentiert sind.

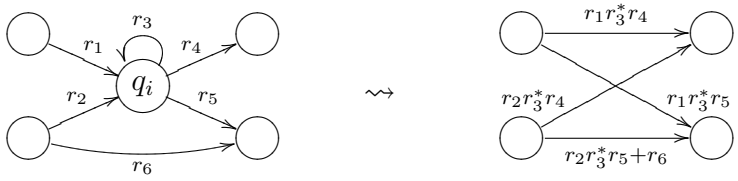


Abbildung 2.11. Elimination des Zustandes q_i.

Achtung: Bei der Elimination von q_i müssen alle Pfade über q_i berücksichtigt werden; insbesondere müssen Schlingen wie in Abb. 2.12 dargestellt berücksichtigt werden.
Am Ende erhalten wir einen verallgemeinerten Automaten mit einer Kante von q_0 nach q_n, die mit einem regulären Ausdruck beschriftet ist (vgl. Abb. 2.13). Dies ist der reguläre Ausdruck, der die vom endlichen Automaten akzeptierte Sprache bezeichnet.

***Achtung:** Es gibt noch einige Feinheiten, die wir in den Übungen noch genauer betrachten. Beispielsweise müssen zwischendurch verschiedene Kanten zwischen denselben Zuständen zusammengefaßt werden; es kann auch der Fall eintreten, daß es am Ende keine Kante von q_0 nach q_n gibt (was dann?).*

□

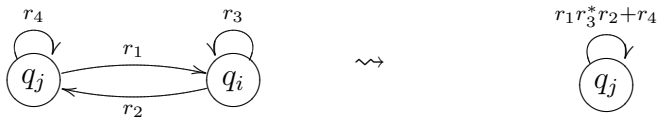


Abbildung 2.12. Ein Sonderfall (eine Schlinge zwischen q_j und q_i).

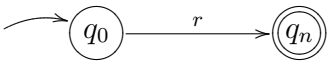


Abbildung 2.13. Der verallgemeinerte Automat am Ende des Eliminationsprozesses.

Bemerkung: Es gibt auch andere Konstruktionen, die dem Warshall-Algorithmus ähneln (siehe Info IV - Skript oder Übungsblatt 6, Aufgabe 2).

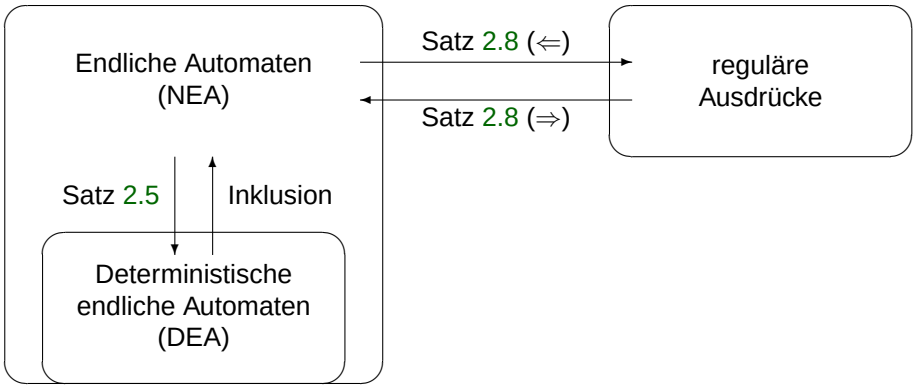
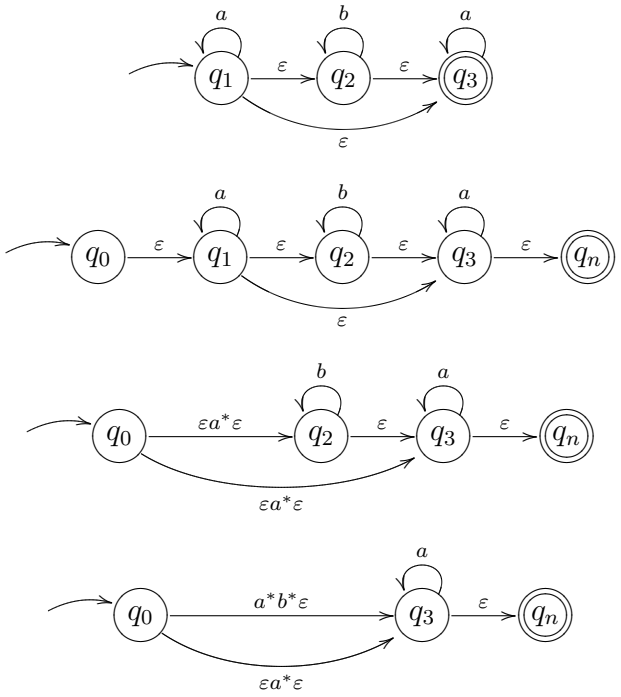


Abbildung 2.14. Überblick über die verschiedenen Repräsentationsmöglichkeiten regulärer Sprachen.

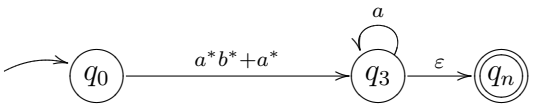
Überblick Insgesamt ergibt sich daraus folgendes Bild für die verschiedenen Repräsentation regulärer Sprachen (vgl. Abb. 2.14):

- Wir kennen nun drei verschiedene Notationen, um reguläre Sprachen syntaktisch zu repräsentieren (später werden wir sogar noch mehr kennenlernen).
- Die Repräsentationen sind effektiv äquivalent, d.h. sie können mechanisch in eine jeweils andere Repräsentation überführt werden. Das wiederum heißt, daß
  - alle Fragen an reguläre Sprachen (im Prinzip) in einer einzigen Repräsentation beantwortet werden können.
  - In der Praxis wählt man sich jeweils die geeigneteste Repräsentation. Will man beispielsweise den Abschluß unter dem Komplement zeigen, wählt man den DEA als Repräsentation. Will man hingegen den Abschluß bezüglich des *-Operators zeigen, so wählt man einen NEA (oder direkt die regulären Ausdrücke, da dort gar nichts zu zeigen ist, da * eine reguläre Operation ist.).

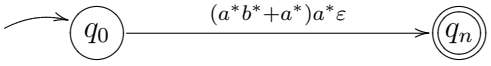
Beispiel 2.4
Ein Beispiel zum Verfahren aus Satz 2.8:



Zusammenfassen von Kanten zwischen den gleichen Knoten.



$a^*b^* + a^*$ könnte man auch zu a^*b^* vereinfachen (einem Rechner müßte man das aber mühsam „beibringen“).



$(a^*b^* + a^*)a^*\varepsilon$ könnte man zu $a^*b^*a^*$ vereinfachen.

Definition 2.9 (Äquivalenz regulärer Ausdrücke) *Zwei reguläre Ausdrücke heißen* äquivalent, *wenn sie dieselbe Sprache bezeichnen.*

Später werden wir ein Entscheidungsverfahren für die Äquivalenz zweier endlicher Automaten kennenlernen. Dann können wir die Äquivalenz regulärer Ausdrücke darauf zurückführen. Davor beschäftigen wir uns aber mit dem Ausrechnen von Äquivalenzen (Abschnitt 2.3).

Folgerung 2.10 *Seien r und s zwei reguläre Ausdrücke, die die Sprachen R bzw. S bezeichnen. Dann existieren effektiv zwei reguläre Ausdrücke, die die Sprachen $\overline{R}$ und $R \cap S$ bezeichnen.*

Beweis: Satz 2.8 und Satz 2.6.2/3. □

2.3 Rechnen mit regulären Ausdrücken

Ein kleiner Ausflug in die "Semantik" bzw. "Logik".

Obwohl man die Äquivalenz regulärer Ausdrücke auf die Äquivalenz von Automaten zurückführen kann, bietet es sich an, die Äquivalenz regulärer Ausdrücke "auszurechnen". Hier lernen wir die Rechenregeln dazu kennen. Diese Rechenregeln reichen aus, um alle Äquivalenzen zu beweisen (Vollständigkeit). Dies werden wir jedoch nicht beweisen.

Leerworteigenschaft: Zuvor führen wir noch einen Hilfsbegriff ein: Die Menge der regulären Ausdrücke, die die *Leerworteigenschaft (LWE)* erfüllen, ist induktiv definiert durch:

- ε erfüllt die Leerworteigenschaft
- $(r + s)$ erfüllt die Leerworteigenschaft, wenn r oder s die Leerworteigenschaft erfüllen.
- (rs) erfüllt die Leerworteigenschaft, wenn r und s die Leerworteigenschaft erfüllen.
- (r^*) erfüllt die Leerworteigenschaft.

Die Idee hinter der Definition der Leerworteigenschaft ist, dass die von einem regulären Ausdruck bezeichnete Sprache genau dann das leere Wort enthält, wenn der reguläre Ausdruck die Leerworteigenschaft besitzt.

2.3.1 Axiome

Wir führen nun die Axiome und danach die Rechenregeln für reguläre Ausdrücke ein. Die Axiome (1)–(9) sind die Axiome nach Salomaa und Urponen. Allerdings kommt bei Salomaa und Urponen ε nicht als regulärer Ausdruck vor. Aus diesem Grund definieren wir in Axiom (0) die Bedeutung von ε explizit. Die Axiome $(0')$–$(0''')$ sind Axiome, die uns das Rechnen etwas zu vereinfachen; diese Axiome braucht man aber nicht, um die Vollständigkeit der Rechenregeln zu beweisen:

$$
\begin{array}{lrcl}
(0) & \varepsilon & = & \varnothing^* \\
(0') & r & = & r \\
(0'') & r + \varnothing & = & r \\
(0''') & r + r & = & r \\
 & & & \\
(1) & (r + s) + t & = & r + (s + t) \\
(2) & (rs)t & = & r(st) \\
(3) & r + s & = & s + r \\
(4) & r(s + t) & = & rs + rt \\
(5) & (r + s)t & = & rt + st \\
(6) & r\varepsilon & = & r \\
(7) & r\varnothing & = & \varnothing \\
(8) & r^* & = & rr^* + \varepsilon \\
(9) & r^* & = & (r + \varepsilon)^*
\end{array}
$$

2.3.2 Rechenregeln

Zunächst die üblichen (aus der Schule bekannten) Rechenregeln:

Kongruenz: Wenn $r = s$ gilt, kann in einem Ausdruck t_1 ein Vorkommen von r durch s ersetzt werden. Für den resultierenden Ausdruck t_2 gilt dann $t_1 = t_2$. Diese Regel wird auch *Substitutionsregel* genannt.

Symmetrie: Wenn $r = s$ gilt, dann gilt auch $s = r$.

Transitivität: Wenn $r = s$ und $s = t$ gilt, dann auch $r = t$.

Nun eine etwas unüblichere Regel, die speziell auf reguläre Ausdrücke zugeschnitten ist:

Gleichungsauflösung: Wenn die Gleichung $r = sr + t$ gilt und s die Leerworteigenschaft *nicht* erfüllt, dann gilt auch die Gleichung $r = s^*t$.

Offensichtlich sind alle Axiome und alle Regeln bis auf die Gleichungsauflösung korrekt. Die Gleichungsauflösungsregel ist natürlich auch korrekt. Das werden wir uns in der Übung (Blatt 3, Aufgabe 4) genauer ansehen. Die Nebenbedingung, daß s die LWE nicht erfüllt, garantiert, daß die Gleichung $r = sr + t$ genau eine Lösung für r hat: $r = s^*t$. Daß $r = s^*t$ eine Lösung ist, sieht man leicht durch Einsetzen; daß sie eindeutig ist (wenn s die LWE erfüllt), muß man sich noch überlegen.

Beispiel 2.5
Als ein Beispiel für das Rechnen mit regulären Ausdrücken beweisen wir die Gleichung $(a + b)^* = a^*(a + b)^*$:

(1)	$(a+b)^*$	$=$	$(a+b)(a+b)^* + \varepsilon$	Axiom (8)
(2)	$a + a$	$=$	a	Axiom (0”’)
(3)	$(\underline{(a+a)} + b)(a+b)^* + \varepsilon$	$=$	$(\underline{a} + b)(a+b)^* + \varepsilon$	Kongr. mit (2)
(4)	$(\underline{a} + b)(a+b)^* + \varepsilon$	$=$	$(\underline{(a+a)} + b)(a+b)^* + \varepsilon$	Sym. mit (3)
(5)	$(a+a) + b$	$=$	$a + (a+b)$	Axiom (1)
(6)	$(\underline{(a+a) + b})(a+b)^* + \varepsilon$	$=$	$(\underline{a + (a+b)})(a+b)^* + \varepsilon$	Kongr. mit (5)
(7)	$\underline{(a + (a+b))(a+b)^*}$	$=$	$a(a+b)^* + (a+b)(a+b)^*$	Axiom (5)
(8)	$\underline{(a + (a+b))(a+b)^* + \varepsilon}$	$=$	$\underline{(a(a+b)^* + (a+b)(a+b)^*) + \varepsilon}$	Kongr. mit (2)
(9)	$(a+b)^*$	$=$	$((a+a) + b)(a+b)^* + \varepsilon$	Trans. (1,4)
(10)	$(a+b)^*$	$=$	$(a + (a+b))(a+b)^* + \varepsilon$	Trans. (9,6)
(11)	$(a+b)^*$	$=$	$(a(a+b)^* + (a+b)(a+b)^*) + \varepsilon$	Trans. (10,8)
(12)	$(a(a+b)^* + (a+b)(a+b)^*) + \varepsilon$	$=$	$a(a+b)^* + ((a+b)(a+b)^* + \varepsilon)$	Axiom (1)
(13)	$(a+b)^*$	$=$	$a(a+b)^* + ((a+b)(a+b)^* + \varepsilon)$	Trans. (11,12)
(14)	$(a+b)^*$	$=$	$a^*((a+b)(a+b)^* + \varepsilon)$	Gleichungsaufl.
(15)	$a^*\underline{(a+b)^*}$	$=$	$a^*\underline{((a+b)(a+b)^* + \varepsilon)}$	Kongr. mit (1)
(16)	$a^*((a+b)\underline{(a+b)^* + \varepsilon})$	$=$	$a^*\underline{(a+b)^*}$	Sym. mit (15)
(17)	$(a+b)^*$	$=$	$a^*(a+b)^*$	Trans.(14,16)

Man sieht, daß das Beweisen durch strikte Regelanwendung und unter ausschließlicher Benutzung der Axiome sehr mühselig ist. Im Prinzip ist es aber möglich, für zwei äquivalente reguläre Ausdrücke die Äquivalenz über die Rechenregeln auszurechnen (das können wir hier aber nicht beweisen)!
In der Praxis, gibt man nur die Beweisidee an (und spart sich die lästigen technische Details). Die Idee hinter obigem Beweis läßt sich relativ knapp formulieren:

$$
\begin{array}{lllll}
(1) & (a+b)^* & = & (a+b)(a+b)^*+\varepsilon & \\
(10) & & = & (a+(a+b))(a+b)^*+\varepsilon & \\
(13) & & = & a(a+b)^*+((a+b)(a+b)^*+\varepsilon) & \\
(14) & (a+b)^* & = & a^*((a+b)(a+b)^*+\varepsilon) & \text{Gleichungsaufl.} \\
(17) & & = & a^*(a+b)^* &
\end{array}
$$

Frage: Liefern uns die Rechenregeln für reguläre Ausdrücke ein Verfahren, um die Äquivalenz zweier Ausdrücke zu entscheiden?

Antwort: Nein! Wir können zwar durch systematisches Durchprobieren aller Herleitungen feststellen, wenn zwei Ausdrücke äquivalent sind (wenn zwei reguläre Ausdrücke äquivalent sind, dann gibt es wegen der Vollständigkeit der Rechenregeln eine Herleitung für den Beweis). Aber wenn die Ausdrücke nicht äquivalent sind, dann würden wir es nie erfahren (siehe Semi-Entscheidbarkeit, Aufzählbarkeit).
Trotzdem ist das Rechnen mit regulären Ausdrücken oft sinnvoll (zum Beispiel, um Ausdrücke zu vereinfachen).

Ein Entscheidungsverfahren für die Äquivalenz zweier regulärer Ausdrücke werden wir in Abschnitt 4 kennen lernen.

3 Eigenschaften regulärer Sprachen

In diesem Abschnitt betrachten wir „Eigenschaften regulärer Sprachen“. Diese Formulierung läßt Raum für veschiedene Interpretationen. Einerseits können damit Eigenschaften gemeint sein, die jede reguläre Sprache besitzen muß. Damit beschäftigen wir uns in Abschnitt 3.1. Andererseits können mit der Formulierung auch die Eigenschaften der gesamten Klasse aller regulären Sprachen gemeint sein. Dazu gehören die Abschlußeigenschaften. Mit den Abschlußeigenschaften regulärer Sprachen werden wir uns in Abschnitt 3.2 beschäftigen.

3.1 Eigenschaften einer regulären Sprache

Motivation: Wie beweist man, dass eine bestimmte Sprache nicht regulär ist? Indem man nachweist, dass sie eine Eigenschaft nicht besitzt, die jede reguläre Sprache besitzen muß. Dazu ist es natürlich hilfreich, Eigenschaften zu kennen, die jede reguläre Sprache besitzen muß.

Beispiel 2.6
Für die Sprache der Wörter, die genausoviele a's wie b's enthalten, d.h. für $L = \{w \,|\, |w|_a = |w|_b\}$ hat die Relation R_L einen unendlichen Index. Wir wissen aber (aus der Übung, vgl. Blatt 2, Aufgabe 4), dass eine Sprache nur dann regulär ist, wenn der Index von R_L endlich ist. Das Ausrechnen des Index R_L einer Sprache ist jedoch etwas mühselig. Also überlegen wir uns zunächst eine praktikablere Eigenschaft.

3.1.1 Pumping-Lemma (uvw-Theorem)

Wenn eine Sprache L regulär ist, wird sie nach den Sätzen 2.5 und 2.8 von einem deterministischen endlichen Automaten $A = (Q, \Sigma, \delta, I, F)$ akzeptiert.

In der folgenden Argumentation reicht es eigentlich, einen buchstabierenden Automaten zu betrachten – der Automat muß nicht deterministisch sein.

Sei $n = |Q|$. Wenn nun L ein Wort x mit $|x| \geq n$ enthält, dann gibt es im Automaten A einen Pfad von einem Startzustand p zu einem Endzustand r, auf dem mindestens ein Zustand q mehrmals vorkommt wie dies in Abb. 2.15 dargestellt ist.

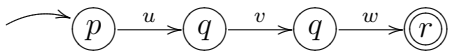


Abbildung 2.15. Pfad des Automaten für ein Wort x mit $|x| \geq n$.

Sei nun q der erste Zustand, der auf dem Pfad doppelt vorkommt. Dann läßt sich x in Teilwörter u,v, w zerlegen (d.h. $x = uvw$), so daß

- $|v| \geq 1$ (da der Automat A buchstabierend ist, kann das Wort v zwischen dem ersten Auftreten von q und dem zweiten Auftreten von q nicht leer sein) und

- $|uv| \leq n$ (da der Automat buchstabierend ist, und q der erste Zustand ist, der doppelt auftritt, können von p bis zum zweiten Zustand q höchstens n Übergänge vorkommen; da der Automat buchstabierend ist, muß also gelten $|uv| \leq n$).

  $u = \varepsilon$ ist möglich.

Den Teilpfad vom ersten Auftreten von q bis zum zweiten Auftreten von q können wir nun aber weglassen oder beliebig oft wiederholen; also werden auch die Wörter uw und uv^iw mit $i \in \mathbb{N} \setminus \{0\}$ vom Automaten akzeptiert. Also ist jedes Wort aus uv^*w auch ein Wort von L.

Lemma 2.11 (Pumping-Lemma) *Für jede reguläre Sprache L gibt es ein $n \in \mathbb{N}$, so dass für jedes $x \in L$ mit $|x| \geq n$ eine Zerlegung $x = uvw$ mit $|v| \geq 1$ und $|uv| \leq n$ existiert, so daß $uv^*w \subseteq L$ gilt.*

Beweis: Ausformulierung unserer Vorüberlegung. □

Bemerkung: Um die Analogie zum Pumping-Lemma für kontextfreie Sprachen, das wir später noch kennenlernen werden, zu verdeutlichen, wird $uv^*w \subseteq L$ meist äquivalent durch $uv^iw \in L$ für alle $i \in \mathbb{N}$ ausgedrückt.

Beispiel 2.7 (Anwendung des Pumping-Lemmas)
Wir beweisen, dass die Sprache der Palindrome L über $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ nicht regulär ist. Dazu nehmen wir an, dass die Palindrome regulär sind und führen dies mit Hilfe des Pumping-Lemmas zum Widerspruch.

Wenn L regulär ist, gibt es ein $n \in \mathbb{N}$, so dass für jedes Wort $x \in L$ eine Zerlegung $x = uvw$ existiert, mit $|uv| \leq n$ und $|v| \geq 1$ und $uv^i w \in L$ für jedes $i \in \mathbb{N}$. Wir wählen

$$x = \overbrace{aa...aa...aaa}^{n} b \overbrace{aa...aa}^{n}$$

Sei nun u, v, w eine beliebige Zerlegung von x mit $|uv| \leq n$ und $|v| \geq 1$:

$$x = \underbrace{aa...a}_{u} \underbrace{a...aa}_{v} \underbrace{abaa...aa}_{w}$$

Also gilt (wegen $|v| \geq 1$):

$$uw = \overbrace{aa...aaa}^{<n} \mathsf{b} \overbrace{aa...aa}^{n}$$

Dies ist jedoch kein Palindrom. Also gilt $uv^0w = uw \notin L$. Es gibt also für x keine Zerlegung wie im Pumping-Lemma gefordert und damit ist die Sprache der Palindrome nicht regulär.

Bemerkung: Es gibt nicht-reguläre Sprachen, für die sich mit Hilfe des Pumping-Lemmas nicht beweisen läßt, daß sie nicht regulär sind.

> *Es gibt daher noch einige Verallgemeinerungen des Pumping-Lemmas. Eine davon werden wir in der Übung (Blatt 4, Aufgabe 1) kennenlernen.*

3.1.2 Endlicher Index von R_L (Minimalautomat)

In der Übung (Blatt 2, Aufgabe 4) haben wir bereits gesehen, dass eine Sprache L, für die die Relation R_L einen unendlichen Index besitzt, von keinem deterministischen endlichen Automaten akzeptiert wird, also nicht regulär ist. Jetzt präzisieren wir diese Aussage.

Zur Erinnerung: Für eine Sprache $L \subseteq \Sigma^*$ definieren wir die Relation R_L durch $u \; R_L \; v$ genau dann, wenn für jedes $x \in \Sigma^*$ gilt $ux \in L$ genau dann, wenn $vx \in L$. Man kann leicht nachweisen, daß R_L eine Äquivalenzrelation ist. Wir können also über den Index von R_L reden.

Satz 2.12 (Myhill, Nerode)
Eine Sprache $L \subseteq \Sigma^*$ ist genau dann regulär, wenn der Index von R_L endlich ist.

Beweis: Wir beweisen die beiden Richtungen der Genau-dann-wenn-Beziehung separat:

„$\Rightarrow$“: Diese Richtung haben wir bereits in Übung 2 (Aufgabe 4a. und d.) gezeigt.

> *In der Übung haben wir die Kontraposition dieser Aussage gezeigt: Wenn R_L unendlichen Index hat, dann gibt es keinen deterministischen und vollständigen endlicher Automaten, der L akzeptiert (d.h. dann ist L nicht regulär).*

Die zentrale Idee dabei ist, dass alle Wörter, die vom Startzustand aus in denselben Zustand führen, gemäß der Relation R_L äquivalent sind.

„⇐“: Sei also der Index von R_L endlich. Wir müssen zeigen, daß L dann regulär ist. Dazu konstruieren wir einen endlichen Automaten $A_L = (Q, \Sigma, \delta, q_0, F)$, der L akzeptiert:

$$\begin{aligned} Q &= \{[u]_{R_L} \mid u \in \Sigma^*\} \qquad \text{d.h. die Zustände sind gerade die Äquivalenzklassen von } R_L \\ \delta &= \{([u], a, [ua]) \mid u \in \Sigma^*, a \in \Sigma\} \\ q_0 &= [\varepsilon] \\ F &= \{[u] \mid u \in L\} \end{aligned}$$

Da der Index von R_L endlich ist, ist auch Q endlich; A_L ist also ein endlicher Automat.
Darüber hinaus ist der Automat A_L deterministisch. Dazu müssen wir berücksichtigen, daß dieselbe Äquivalenzklasse verschiedene Repräsentanten u und v besitzen kann: $[u] = [v]$. Wir müssen also zeigen, daß für beide Repräsentanten der in δ definierte Übergang für $a \in \Sigma$ in dieselbe Äquivalenzklasse führt. Gemäß Definition von δ gilt $([u], a, [ua]) \in \delta$ und $([v], a, [va]) \in \delta$. Wir müssen also zeigen $[ua] = [va]$ (d.h. $ua\ R_L\ va$):
Wegen $[u] = [v]$ gilt $u\ R_L\ v$. Gemäß Definition von R_L gilt dann für jedes $x \in \Sigma^*$: $ux \in L$ genau dann wenn $vx \in L$. Insbesondere gilt für alle $y \in \Sigma^*$. $uay \in L$ genau dann, wenn $vay \in L$ gilt. Also gilt $ua\ R_L\ va$. Also gilt $[ua] = [va]$ und damit ist der Automat A_L deterministisch.

> *Hier haben wir gezeigt, daß R_L eine sog. Rechtskongruenz (bzgl. der Konkatenation) ist. Dies ist genau die Eigenschaft von R_L, die gewährleistet, daß δ deterministisch ist.*

Außerdem ist A_L vollständig, denn für jedes $[u] \in Q$ gilt per Definition von δ unmittelbar $([u], a, [ua]) \in \delta$. Durch Induktion über die Länge der Wörter kann man nun zeigen, daß für jedes Wort $w \in \Sigma^*$ gilt $([\varepsilon], w, [w]) \in \delta^*$. Insgesamt gilt:

$$\begin{aligned} w \in L &\Leftrightarrow [w] \in F \\ &\Leftrightarrow ([\varepsilon], w, [w]) \in \delta^* \text{ und } [w] \in F \\ &\Leftrightarrow w \in L(A_L) \end{aligned}$$

Also gilt $L(A_L) = L$ und damit ist L regulär. □

Beobachtungen:

1. Für den Automaten A_L aus dem Beweis von Satz 2.12 gilt auch $L(A_L) = L$, wenn der Index von R_L nicht endlich ist. Allerdings ist der Automat A_L dann nicht endlich.

   > *Für jede Sprache gibt es einen unendlichen Automaten A mit $L(A) = L$; z.B. den Automaten A_L.*

2. Es gibt keinen deterministischen und vollständigen Automaten der weniger Zustände hat als der Index von R_L. Der Automat ist also ein minimaler deterministischer und vollständiger Automat für L. A_L heißt deshalb auch der *Minimalautomat* für L.

3. Man kann sich leicht überlegen, dass jeder Automat, der genausoviele Zustände wie A_L besitzt und L akzeptiert, zu A_L isomorph ist. Der Minimalautomat ist also bis auf Isomorphie eindeutig.

   > *Achtung: Das gilt für nicht-deterministische endliche Automaten nicht.*

Frage: Wie findet man für einem endlichen Automaten den äquivalenten Minimalautomaten?

Die explizite Konstruktion über R_L und A_L ist nicht praktikabel.

Idee: Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer und vollständiger endlicher Automat, der o.B.d.A. keine unerreichbaren Zustände besitzt.

Die unerreichbaren Zustände kann man gegebenenfalls einfach streichen, da sie keinen Einfluß auf die akzeptierte Sprache des Automaten haben.

Wir nennen zwei Zustände $p, q \in Q$ *äquivalent* (in Zeichen $p \equiv_A q$), wenn für jedes $w \in \Sigma^*$ mit $(p, w, r) \in \delta^*$ und $(q, w, r') \in \delta^*$ gilt: $r, r' \in F$ oder $r, r' \notin F$. Äquivalente Zustände eines Automaten können wir identifizieren (verschmelzen), ohne die akzeptierte Sprache des Automaten zu verändern: Das Ergebnis ist dann der Minimalautomat.
Wir müssen also nur noch die Menge der äquivalenten Zustände berechnen. Wir gehen dazu umgekehrt vor und bestimmen iterativ alle Paare von nicht-äquivalenten Zuständen ($p \not\equiv q$) mit Hilfe der folgenden Regeln:

- $p \in F \wedge q \notin F \qquad \Rightarrow \qquad p \not\equiv q$

  Wenn von zwei Zuständen einer ein Endzustand ist und der der andere nicht, dann sind die beiden Zustände nicht äquivalent.

- $(p, a, r) \in \delta \wedge (q, a, r') \in \delta \wedge r \not\equiv r' \qquad \Rightarrow \qquad p \not\equiv q$

  Wenn von zwei Zuständen ein a-Übergang zu zwei nicht-äquivalenten Zuständen existiert, dann sind auch die beiden Zustände selbst nicht äquivalent.

  Diese Regel wenden wir so lange an, bis aufgrund der Regel keine neuen Paare $p \not\equiv q$ mehr hinzugefügt werden können. Die dann verbleibenden Paare von Zuständen sind jeweils äquivalent.

Dieses Verfahren kann man effizient implementieren (Zeitaufwand: $O(|\Sigma| \cdot |Q^2|)$).

Ein Beispiel hierzu findet sich auf Folie 48.

Redaktioneller Hinweis: Das Beispiel von Folie 48 sollte irgendwann ins Skript übernommen werden.

3.2 Weitere Abschlußeigenschaften

Wir haben bereits gesehen, das die Klasse der regulären Sprachen unter den Operationen $\cup$, $\cap$, $\neg$, $\circ$ und * abgeschlossen ist (siehe Abschnitt 1.4, Satz 2.6 auf Seite 22). Nun werden wir weitere Operationen behandeln.

3.2.1 Die Operationen

Zunächst definieren wir einige weitere Operationen auf Sprachen, bezüglich derer wir dann die Abgeschlossenheit der regulären Sprachen (und später noch weiterer Sprachklassen) untersuchen.

Homomorphismen und inverse Homomorphismen Seien Σ und Δ (nicht notwendigerweise disjunkte) Alphabete und $h : \Sigma \to \Delta^*$ eine Abbildung. Die Abbildung h läßt sich kanonisch zu einem *Homomorphismus*: $\overline{h} : \Sigma^* \to \Delta^*$ erweitern:

$$\left.\begin{array}{rcl} \overline{h}(\varepsilon) & = & \varepsilon \\ \overline{h}(a) & = & h(a) \\ \overline{h}(aw) & = & h(a)\overline{h}(w) \end{array}\right\} \text{zeichenweise Anwendung der Abbildung}$$

Den Strich (Balken) über h benötigen wir nur zur Unterscheidung der gegebenen Abbildung h und des durch sie definierten Homomorphismus $\overline{h}$. Dieser Unterschied geht aber meist aus dem Kontext hervor. Deshalb wird der Strich über h meist weggelassen – wir werden ihn ab jetzt auch weglassen.

Für eine Sprache $L \subseteq \Sigma^*$ definieren wir $h(L) = \{h(w) \mid w \in L\} \subseteq \Delta^*$.

Für eine Sprache $L \subseteq \Delta^*$ definieren wir $h^{-1}(L) = \{x \in \Sigma^* \mid \overbrace{h(x) = w, w \in L}^{h(x)\in L}\}$.
Die Sprache $h(L)$ heißt *(homomorphes) Bild* von L unter h; wir sagen auch $h(L)$ entsteht durch Anwendung eines Homomorphismus aus L. Die Sprache $h^{-1}(L)$ heißt *Urbild* von L unter h; wir sagen auch $h^{-1}(L)$ entsteht durch Anwendung eines inversen Homomorphismus auf L.

Gemäß Definition gilt für ein Wort x genau dann $x \in h^{-1}(L)$, wenn $h(x) \in L$ gilt. Dies scheint trivial und ist es auch. Dennoch sollten Sie diese Äquivalenz verinnerlichen, da sie in vielen Beweisen der „Schlüssel zum Erfolg“ ist.

Beispiel 2.8
Sei $\Sigma = \{a, b\}$ ein Alphabet, sei $h : \Sigma \to \Sigma^*$ eine Abbildung mit $h(a) = a$ und $h(b) = a$ und sei $L = \{ab, abab, ababab, \dots\}$ eine Sprache über Σ^*. Dann ist $h(L) = \{aa, aaaa, aaaaaa, \dots\}$ das homomorphe Bild von L unter h.
Sei nun $\Delta = \{c\}$ ein weiteres Alphabet und sei $g : \Delta \to \Sigma^*$ eine Abbildung mit $g(c) = ab$. Dann ist $g^{-1}(L) = \{c, cc, ccc, \dots\}$ das Urbild von L unter g.

Substitution Sei $f : \Sigma \to 2^{\Delta^*}$ eine Abbildung, so dass für jedes $a \in \Sigma$ die Sprache (!) $f(a)$ regulär ist. Die Abbildung läßt sich kanonisch zu einer Abbildung $\overline{f} : \Sigma^* \to 2^{\Delta^*}$ erweitern, die wir *Substitution* nennen:

$$\left.\begin{array}{rcl} \overline{f}(\varepsilon) & = & \{\varepsilon\} \\ \overline{f}(aw) & = & f(a)\overline{f}(w) \end{array}\right\} \begin{array}{l}\text{zeichenweise Anwendung von } f \text{ ordnet} \\ \text{einem Wort eine Sprache zu.}\end{array}$$

Für $L \subseteq \Sigma^*$ definieren wir $\overline{f}(L) = \bigcup_{w\in L} \overline{f}(w)$. Wir sagen, die Sprache $f(L)$ entsteht aus L durch Substitution.

Wie bei Homomorphismen lassen wir den Überstrich bei $\overline{f}$ in Zukunft weg.

Beispiel 2.9
Sei $\Sigma = \{a, b\}$ ein Alphabet, $f : \Sigma \to 2^{\Sigma^*}$ eine Abbildung mit $f(a) = a^* = \{\varepsilon, a, aa, aaa, \dots\}$ und $f(b) = b^* = \{\varepsilon, b, bb, bbb, \dots\}$. Dann gilt $f(aba) = a^*b^*a^*$, $f(\Sigma^*) = \Sigma^*$ und $f(aa) = a^*$.

Quotientenbildung Seien $L_1, L_2 \subseteq \Sigma^*$ zwei beliebige Sprachen. Dann ist der *Quotient* von L_1 und L_2 (in Zeichen L_1/L_2) definiert durch $L_1/L_2 = \{u \in \Sigma^* \mid v \in L_2, uv \in L_1\}$.

Beispiel 2.10
Seien $L_1 = a^*ba^*$, $L_2 = ba^*b$ und $L_3 = a^*b$ drei Sprachen. Dann gilt: $L_1/L_2 = \varnothing$, $L_1/L_3 = a^*$ und $L_2/L_3 = ba^*$.

- *Wir haben den Quotienten direkt auf den Sprachen definiert. Man könnte den Quotienten auch auf Wörtern definieren und dann auf die Sprachen übertragen, wie wir dies beispielsweise für die Konkatenation getan haben. Das Problem dabei ist, daß der Quotient auf Wörtern eine partielle Operation ist. Deshalb haben wir auf diese Definition verzichtet.*

  Der Quotient auf Wörtern ist die inverse Operation zur Konkatenation: $(u \circ v)/v = u$. Wenn man die Konkatenation als Multiplikation liest, dann ist der Quotient die Division – daher der Name „Quotient“.

- *Wir haben den sog.* Rechtsquotienten *definiert. Es gibt analog dazu auch noch den* Linksquotienten $(u \circ v) \setminus u = v$.

  Dann muß man aber sehr aufpassen, damit man den Linksquotienten auf Sprachen nicht mit der Mengendifferenz verwechselt. Deshalb verzichten wir vollständig auf die Definition des Linksquotienten. Sollten wir den Linksquotienten als Operation benötigen, können wir uns diese Operation aus dem Rechtsquotienten und der Spiegelung „zusammenbasteln".

Spiegelsprache Für $w \in \Sigma^*$ mit $w = a_1 \dots a_n$ und $a_i \in \Sigma$ bezeichnet $\overleftarrow{w} = a_n a_{n-1} \dots a_1$ das rückwärts gelesene Wort w; wir nennen $\overleftarrow{w}$ das *Spiegelwort* von w.
Die *Spiegelsprache* $\overleftarrow{L}$ einer Sprache L ist dann wie folgt definiert: $\overleftarrow{L} = \{\overleftarrow{w} \mid w \in L\}$.

3.2.2 Die Abschlußeigenschaften

Satz 2.13 (Abschlußeigenschaften) *Die regulären Sprachen sind abgeschlossen unter:*

- Homomorphismen, *d.h. für eine reguläre Sprache $L \subseteq \Sigma^*$ und einen Homomorphismus h ist auch $h(L)$ regulär.*
- Inversen Homomorphismen, *d.h. für eine reguläre Sprache $L \subseteq \Sigma^*$ und einen Homomorphismus h ist auch $h^{-1}(L)$ regulär.*
- Substitution, *d.h. für eine reguläre Sprache $L \subseteq \Sigma^*$ und eine Substitution f ist auch $f(L)$ regulär.*
- Quotientenbildung *mit einer beliebigen Sprache L', d.h. für eine reguläre Sprache $L \subseteq \Sigma^*$ und eine beliebige Sprache $L' \subseteq \Sigma^*$ ist auch der Quotient L/L' regulär.*
- Spiegelung, *d.h. für eine reguläre Sprache $L \subseteq \Sigma^*$ ist auch ihre Spiegelsprache $\overleftarrow{L}$ regulär.*

Sowohl für Homomorphismen, inversen Homomorphismen, Substitution als auch für die Spiegelung existieren effektive Verfahren, um aus einem endlichen Automaten, der die Ausgangssprache akzeptiert, einen Automaten zu konstruieren, der die Sprache nach Anwendung der entsprechenden Operation akzeptiert. Wir sagen deshalb, daß die regulären Sprachen unter diesen Operationen effektiv abgeschlossen *sind.*

Im ersten Augenblick ist es vielleicht etwas überraschend, daß reguläre Sprachen unter Quotientenbildung mit **beliebigen** *Sprachen abgeschlossen sind (versuchen Sie das mal mit Vereinigung oder Durchschnitt). Ein Blick in den Beweis offenbart dann jedoch, daß dies doch nicht so verwunderlich ist. Die regulären Sprachen sind zwar unter Quotientenbildung mit beliebigen Sprachen abgeschlossen, aber nicht effektiv (das sollte uns nicht so sehr überraschen). Wir werden jedoch noch sehen, daß reguläre Sprachen effektiv unter Quotientenbildung abgeschlossen sind, wenn wir uns auf Quotientenbildung mit regulären Sprachen beschränken.*

Beweis:
Wir beweisen nun die Abschlußeigenschaften der regulären Sprachen für jede Operation separat.

Substitution Sei L regulär und $f : \Sigma \to 2^{\Delta^*}$ eine Substitution. O.B.d.A. existiert dann ein regulärer Ausdruck r, der die Sprache L bezeichnet und für jedes $a \in \Sigma$ existiert ein regulärer Ausdruck r_a, der die Sprache $f(a)$ bezeichnet (da per Definition $f(a)$ für jedes $a \in \Sigma$ regulär ist).
Wir ersetzen nun in r jedes Auftreten eines Zeichen a durch r_a. Dazu definieren eine Abbildung $\tilde{f}$ von der Menge der regulären Ausdrücke über Σ in die Menge der regulären Ausdrücke über Δ, die diese Ersetzung durchführt. $\tilde{f}(r)$ liefert uns dann den gewünschten regulären Ausdruck. Die Abbildung $\tilde{f}$ können wir induktiv über den Aufbau der regulären Ausdrücke über Σ wie folgt definieren:

$$
\begin{array}{lcll}
\tilde{f}(\varnothing) & = & \varnothing & \\
\tilde{f}(\varepsilon) & = & \varepsilon & \\
\tilde{f}(a) & = & r_a & \text{Hier passiert's : } a \text{ wird im regulären Ausdruck durch } r_a \text{ ersetzt.} \\
\tilde{f}(s+t) & = & \tilde{f}(s) + \tilde{f}(t) & \\
\tilde{f}(st) & = & \tilde{f}(s)\tilde{f}(t) & \\
\tilde{f}(s^*) & = & \tilde{f}(s)^* &
\end{array}
$$

Man kann nun induktiv über den Aufbau der regulären Ausdrücke zeigen, dass $\tilde{f}(r)$ die Sprache $f(L)$ bezeichnet, wenn L die von r bezeichnete Sprache ist.

Wichtiger als die Erkenntnis, daß reguläre Sprachen unter Substitution abgeschlossen sind, ist die benutzte Beweistechnik zum Nachweis des Abschlusses unter Substitution: Wir führen die Substitution auf der syntaktischen Repräsentation der Sprache durch. Hier sind es die regulären Ausdrücke; man könnte aber auch die Grammatiken benutzen (vgl. das entsprechende Ergebnis für kontextfreie Sprachen).

Homomorphismen Da ein Homomorphismus als eine spezielle Substitution aufgefaßt werden kann, ist nichts mehr zu zeigen.

Inverser Homomorphismus Sei L regulär und sei $h : \Sigma^* \to \Delta^*$ ein Homomorphismus. O.B.d.A. existiert ein deterministischer und vollständiger endlicher Automat $A = (Q, \Sigma, \delta, q_0, F)$, der L akzeptiert.
Wir konstruieren nun aus A einen Automaten A', der $L' = h^{-1}(L)$ akzeptiert. Dabei machen wir uns (wie angekündigt) zunutze, daß $w \in h^{-1}(L)$ genau dann gilt, wenn $h(w) \in L$ gilt.

Idee: Wir simulieren also den ursprünglichen Automaten. Wenn wir ein Zeichen a_i lesen, muß der neue Automat den Übergang durchführen, den der ursprüngliche Automat auf dem Wort $h(a_i)$ durchführen würde (wir simulieren A auf $h(w)$ bzw. auf $h(a_i)$):

$$\begin{array}{rcccccccl} w & = & a_1 & a_2 & a_3 & \dots & a_n & & \in h^{-1}(L) \\ & & \Downarrow & \Downarrow & \Downarrow & & \Downarrow & & \text{gdw.} \\ h(w) & = & \underset{h(a_1)}{w_1} & \underset{h(a_2)}{w_2} & \underset{h(a_3)}{w_3} & \dots & \underset{h(a_n)}{w_n} & & \in L \\ & & & & & & & & \text{gdw.} \end{array}$$

$$q_0 \xrightarrow{w_1}{}^* q_1 \xrightarrow{w_2}{}^* q_2 \xrightarrow{w_3}{}^* \dots \xrightarrow{w_n}{}^* q_n \wedge q_n \in F$$

Die „Simulation“ des Übergangs von A auf $h(a)$ erreichen wir, indem wir in A' einen Übergang $(p, a, q) \in \delta'$ ausführen, wenn für A gilt $(p, h(a), q) \in \delta^*$. Die Zustandsmenge und die Start- und Endzustände bleiben in A' unverändert. Formal definieren wir: $A' = (Q, \Sigma, \delta', q_0, F)$ mit $\delta' = \{(p, a, q) \mid a \in \Delta, (p, \underbrace{h(a)}_{!}, q) \in \delta^*\}$.

***Bemerkung**: Da A deterministisch und vollständig ist, ist auch A' deterministisch und vollständig.*

Man kann sich nun leicht überlegen, daß gilt $w \in L(A')$ gdw. $h(w) \in L(A)$ gdw. $w \in h^{-1}(L)$. A' ist also ein Automat, der $h^{-1}(L)$ akzeptiert.

Genau genommen müßte man durch Induktion über $|x|$ für alle $x \in \Delta^*$ zeigen: $(p, x, q) \in \delta'^*$ genau dann, wenn $(p, h(x), q) \in \delta^*$.

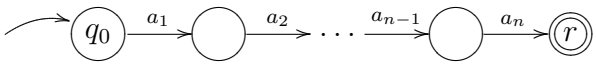
Wieder gilt: Wichtiger als die Erkenntnis, daß reguläre Sprachen unter inversen Homomorphismen abgeschlossen sind, ist die benutzte Beweistechnik: Wir benutzen den akzeptierenden Automaten der ursprünglichen Sprache und „simulieren“ ihn auf der Eingabe $h(w)$.
Der Nachweis, daß einer Sprachklasse unter inversen Homomorphismen abgeschlossen ist, erfolgt in den allermeisten Fällen über das Maschinenmodell mit obiger Simulation.

Quotientenbildung Sei L_1 eine reguläre Sprache und $L_2 \subseteq \Sigma^*$ eine beliebige Sprache. O.B.d.A. existiert ein deterministischer und vollständiger endlicher Automat $A = (Q, \Sigma, \delta, q_0, F)$, der L_1 akzeptiert. Wir konstruieren nun aus A einen Automaten A', der L_1/L_2 akzeptiert
Idee:

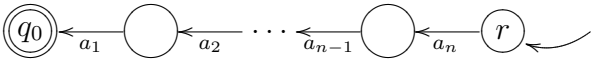
$$\begin{array}{ll} & w \in L_1/L_2 \\ \Leftrightarrow & wv \in L_1, v \in L_2 \\ \Leftrightarrow & q_0 \xrightarrow{wv}{}^* p, v \in L_2, p \in F \\ \Leftrightarrow & q_0 \xrightarrow{w}{}^* q \xrightarrow{v}{}^* p, v \in L_2, p \in F \end{array}$$

Damit nun w vom Automaten A' akzeptiert wird, müssen wir q zu einem Endzustand machen. Wir definieren also $A' = (Q, \Sigma, \delta, q_0, F')$ mit $F' = \{q \in Q \mid (q, v, r) \in \delta^*, v \in L_2, r \in F\}$. Offensichtlich gilt nun $L(A') = L_1/L_2$.

Spiegelung
Idee: Vertausche die Start- und Endzustände miteinander und drehe die Kanten der Übergangsrelation um. Aus einem Automaten:



konstruiert wir also den Automaten:



Achtung: Bei nicht alphabetischen Automaten müssen die Wörter an den Kanten umgedreht werden. Wir können aber ohne Beschränkung der Allgemeinheit annehmen, dass der Automat alphabetisch ist. □

Bemerkung: Alle Konstruktionen bis auf die Quotientenbildung sind effektiv. Wir werden in der Übung (Blatt 5, Aufgabe 1c.) beweisen, dass L_1/L_2 effektiv existiert, wenn auch L_2 regulär ist.

3.2.3 Anwendung der Abschlußeigenschaften

Die Abschlußeigenschaften von regulären Sprachen können wir nun anwenden, um nachzuweisen, daß bestimmte Sprachen regulär sind. Dazu betrachten wir ein Beispiel.

Beispiel 2.11
Für jede reguläre Sprache L ist die Sprache L', in der aus jedem Wort der Sprache L jeder zweite Buchstabe gestrichen wird, auch regulär.
Beweis:
Wir definieren ein Alphabet $\Sigma' = \{a', b', c', \dots\}$, in dem jedes Zeichen aus Σ mit einem Strich versehen ist. Außerdem definieren wir $h : \Sigma \cup \Sigma' \to \Sigma^*$ mit $h(x) = x$ für $x \in \Sigma$ und $h(x') = x$ für $x' \in \Sigma'$.

- Dann ist nach Satz 2.13 die Sprache $h^{-1}(L)$ regulär. Die Sprache $h^{-1}(L)$ enthält alle Wörter aus L, wobei die Buchstaben an beliebigen Stellen mit Strichen versehen werden können.

- Dann ist nach Satz 2.6 die Sprache $h^{-1}(L) \cap ((\Sigma\Sigma')^* \cup (\Sigma\Sigma')^*\Sigma)$ ebenfalls regulär. Diese Sprache enthält genau die Wörter aus L, wobei jedes zweite Zeichen mit einem Strich versehen ist.

- Nun definieren wir $g : \Sigma \cup \Sigma' \to \Sigma^*$ mit mit $g(x) = x$ für $x \in \Sigma$ und $g(x') = \varepsilon$ für $x' \in \Sigma'$. Der Homomorphismus g löscht also genau die Buchstaben, die mit einem Strich versehen sind.

- Dann ist nach Satz 2.13 die Sprache $L' = g(h^{-1}(L) \cap ((\Sigma\Sigma')^* \cup (\Sigma\Sigma')^*\Sigma)$ regulär. Die Sprache L' enthält also genau alle Wörter aus L, wobei in jedem Wort jeder mit einem Strich versehene Buchstabe (d.h. jeder zweite Buchstabe) gelöscht wurde.

□

Durch die Konstruktion einer Sprache L' aus L unter ausschließlicher Verwendung von Operationen unter denen die regulären Sprachen abgeschlossen sind, wissen wir, daß L' regulär ist, wenn L regulär ist. Wir müssen also keinen Automaten mehr für L' konstruieren. Da reguläre Sprachen unter den im Beispiel angewendeten Operationen effektiv abgeschlossen sind, existiert der Automat für L' jedoch effektiv (ohne daß wir eine explizite Konstruktion angeben müssen).

4 Entscheidungsverfahren für reguläre Sprachen

In diesem Abschnitt beschäftigen wir uns damit, welche Fragen bzw. Probleme für reguläre Sprachen entscheidbar sind. Tatsächlich sind die meisten Probleme für reguläre Sprachen entscheidbar.
Zunächst stellen wir uns eine Liste der typischen *Probleme* zusammen:

- Wortproblem: $w \in L$?

  Das Wortproblem brauchen wir uns für reguläre Sprachen nicht anzusehen, da der zugehörige deterministische endliche Automat das Entscheidungsverfahren liefert. Der Vollständigkeit (und Systematik) halber führen wir das Wortproblem hier trotzdem auf.

- Endlichkeit/Unendlichkeit: $|L| < \omega$ oder $|L| = \omega$?
- Leerheit: $L = \varnothing$?
- Äquivalenz: $L_1 = L_2$?
- Inklusion: $L_1 \subseteq L_2$?
- Disjunktheit: $L_1 \cap L_2 = \varnothing$?

  Da reguläre Sprachen effektiv unter Durchschnitt abgeschlossen sind, ist das Disjunktheitsproblem (für reguläre Sprachen) äquivalent zum Leerheitsproblem. Wir werden es also im folgenden nicht mehr gesondert betrachten.

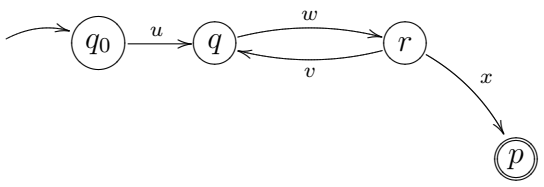
Dabei gehen wir davon aus, daß L (bzw. L_1 und L_2) durch reguläre Ausdrücke oder endliche Automaten gegeben sind. Sollte die Sprache in Form einer Turing-Maschine gegeben sein, können wir damit fast nichts mehr entscheiden (siehe hierzu auch 5).

Satz 2.14 (Entscheidbarkeit von Leerheit und (Un)endlichkeit) *Für jede reguläre Sprache (repräsentiert durch einen regulären Ausdruck oder einen endlichen Automaten) ist entscheidbar, ob $L = \varnothing$ gilt und ob $|L| = \omega$ gilt.*

Beweis: Ohne Beschränkung der Allgemeinheit (siehe Satz 2.5 und 2.8) sei L durch einen deterministischen endlichen Automaten A repräsentiert (d.h. $L = L(A)$).

$L = \varnothing$**:** Die vom Automaten A akzeptierte Sprache L ist genau dann leer, wenn im Automaten kein Pfad von einem Start- zu einem Endzustand existiert. Diese Eigenschaft des Automaten ist offensichtlich entscheidbar. Also ist $L = \varnothing$ für reguläre Sprachen entscheidbar.

$|L| = \omega$**:** Die vom Automaten A akzeptierte Sprache L ist genau dann unendlich, wenn vom Startzustand ein Zyklus erreichbar ist und von irgendeinem Zustand auf diesem Zyklus aus ein Endzustand erreichbar ist:



Auch diese Eigenschaft des Automaten ist offensichtlich entscheidbar. Also ist $|L| = \omega$ entscheidbar.

Man kann auch das Pumping-Lemma benutzen. Dann wird das Verfahren aber sehr ineffizient!

Redaktioneller Hinweis: Die Abbildung kann noch verbessert werden.

□

Bevor wir uns mit dem Äquivalenzproblem und dem Inklusionsproblem beschäftigen, vergegenwärtigen wir uns einige einfache Zusammenhänge aus der Mengenlehre:

$$L_1 \subseteq L_2 \text{ gdw. } L_1 \setminus L_2 = L_1 \cap \overline{L_2} = \varnothing$$

$$L_1 = L_2 \text{ gdw. } L_1 \subseteq L_2 \text{ und } L_2 \subseteq L_1$$

Folgerung 2.15 (Entscheidbarkeit von Äquivalenz und Inklusion)
Für zwei reguläre Sprachen L_1 und L_2 (repräsentiert durch einen regulären Ausdruck oder einen endlichen Automaten) ist entscheidbar, ob $L_1 = L_2$ und ob $L_1 \subseteq L_2$ gilt.

Beweis: Mit der Vorüberlegung über die Zusammenhänge aus der Mengenlehre können wir die Inklusion $L_1 \subseteq L_2$ auf die Frage der Leerheit von $L_1 \cap \overline{L_2}$ zurückführen. Da reguläre Sprachen effektiv unter Durchschnitt und Komplement abgeschlossen sind (Satz 2.6), ist auch $L_1 \cap \overline{L_2}$ regulär und wir können einen endlichen Automaten für diese Sprache konstruieren. Also ist die Leerheit von $L_1 \cap \overline{L_2}$ mit Satz 2.14 entscheidbar. Damit ist $L_1 \subseteq L_2$ entscheidbar.
Zur Entscheidung der Äquivalenz müssen wir nur die beiden Inklusionen $L_1 \subseteq L_2$ und $L_2 \subseteq L_3$ entscheiden.

Alternativ könnte man zur Entscheidung der Äquivalenz zweier regulärer Sprachen auch die zugehörigen Minimalautomaten konstruieren und auf Isomorphie überprüfen.

□

Bemerkung: Im Augenblick ist es vielleicht noch etwas verwunderlich, dass wir so viel Wind um die Abschlußeigenschaften und die Entscheidbarkeit von Leerheit, Unendlichkeit, Inklusion und Äquivalenz machen. Wir werden aber sehen, dass viele dieser Aussagen für andere Sprachklassen nicht gelten. Beispielsweise ist die Disjunktheit zweier kontextfreier Sprachen nicht entscheidbar und ist das Wortproblem für Typ-0-Sprachen (d.h. die rekursiv aufzählbaren Sprachen) nicht entscheidbar. Das ist aber noch Zukunftsmusik.

5 Äquivalente Charakterisierungen regulärer Sprachen

Bisher haben wir im wesentlichen drei Charakterisierungen regulärer Sprachen kennengelernt:

- endliche Automaten (NEA)
- deterministische und vollständige endliche Automaten (DEA)
- reguläre Ausdrücke

Es gibt unzählige weitere Charakterisierungen. Wir betrachten hier stellvertretend die Folgenden:

- erkennbare Mengen
- Gleichungssysteme
- lokale Menge + Homomorphismen
- Zweiwegautomat
- reguläre Grammatiken (Typ-3-Grammatiken)

Die ersten drei Charakterisierungen werden in den Übungen genauer betrachtet (Blatt 4–6). In der Vorlesung gehen wir auf die Zweiwegautomaten und die regulären Grammatiken ein.

5.1 Zweiwegautomaten

Motivation: Warum soll ein Automat eine Zeichenreihe ausschließlich von links nach rechts abarbeiten? Der Zweiwegautomat erlaubt es uns, auch von rechts nach links zu gehen (wenn wir es denn für sinnvoll halten).
Problem: In der Formulierung der endlichen Automaten haben wir gelesene Zeichen "weggeworfen". Das dürfen wir jetzt nicht mehr, denn der Automat kann sich in der Eingabe wieder nach links bewegen. Wir müssen deshalb über Konfigurationen reden. In Abb. 2.16 ist eine *Konfiguration* dargestellt; sie besteht aus dem Eingabewort $a_1 a_2 \dots a_n$, dem aktuellen Zustand q und der aktuellen Position (ähnlich wie später bei Turing-Maschinen).

$$\begin{array}{ccccccc} a_1 & a_2 & \dots & a_i & \dots & a_n \\ & & & \Uparrow & & \\ & & & q & & \end{array}$$

Abbildung 2.16. Eine Konfiguration: Der „Lesekopf" steht auf dem i.-ten Zeichen.

Wenn das Eingabealphabet Σ und die Zustandsmenge Q disjunkt sind, können wir die Konfiguration eindeutig durch $a_1 a_2 \dots a_{i-1} q a_i \dots a_n$ repräsentieren; die Kopfposition i ist durch die Position des Zustands q vor a_i eindeutig bestimmt.
In der Übergangsrelation eines Zweiwegautomaten muß nun auch angeben werden, ob sich der Automat in der Eingabe nach links oder nach rechts bewegt. Der Einfachheit halber betrachten

wir hier nur den deterministischen und vollständigen Zweiwegautomaten, deshalb reicht eine *Übergangsfunktion*:

$$\delta : (Q \times \Sigma) \to (Q \times \{l, r\})$$

Die erste Komponente gibt – wie bei endlichen Automaten – den Folgezustand an, die zweite Komponente die Bewegungsrichtung (l für links, r für rechts).

Definition 2.16 (Zweiwegautomat)
Ein (deterministischer endlicher) Zweiwegautomat *über Σ besteht aus*

- *einer endlichen Menge Q (mit $Q \cap \Sigma = \varnothing$) von* Zuständen,
- *einem* Startzustand $q_0 \in Q$
- *einer* Übergangsfunktion $\delta : (Q \times \Sigma) \to (Q \times \{l, r\})$ *und*
- *einer Menge $F \subseteq Q$ von* Endzuständen.

Wir schreiben $A = (Q, \Sigma, \delta, q_0, F)$.
Ein Wort $\gamma \in \Sigma^* Q \Sigma^*$ heißt Konfiguration *des Automaten A. Für A definieren wir die binäre Relation $\vdash_A$ auf den Konfigurationen durch:*

$$upav \vdash_A uaqv \quad \textit{falls} \quad u, v \in \Sigma^*, a \in \Sigma \textit{ und } \delta(p, a) = (q, r)$$

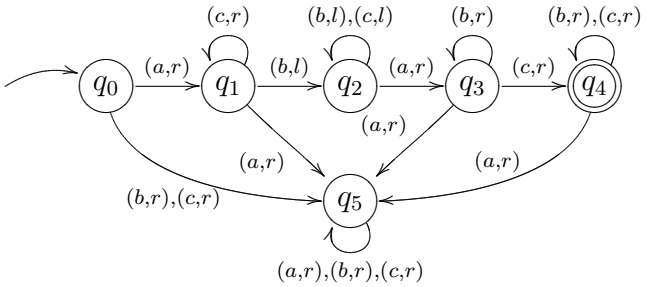
und

$$ubpav \vdash_A uqbav \quad \textit{falls} \quad u, v \in \Sigma^*, a, b \in \Sigma \textit{ und } \delta(p, a) = (q, l)$$

Gemäß der Definition von $\vdash_A$ ist eine Rechtsbewegung nur möglich, wenn rechts von der Kopfposition noch ein Eingabezeichen steht. Eine Linksbewegung ist nur möglich, wenn rechts und links von der Kopfposition noch ein Eingabezeichen steht.

Für zwei Konfigurationen γ und γ' mit $\gamma \vdash_A \gamma'$ nennen wir γ' die Nachfolgekonfiguration *von γ.*
Der Zweiwegautomat A akzeptiert *ein Wort $w \in \Sigma^*$, wenn ein Endzustand $q \in F$ mit $q_0 w \vdash_A^* wq$ existiert. Die Menge aller von A akzeptierten Wörter bezeichnen wir mit $L(A)$.*

Beispiel 2.12
Es ist schwer eine Sprache zu finden, bei der die Charakterisierung über einen Zweiwegautomaten naheliegender erscheint als über einen endlichen Automaten. Hier betrachten wir die Menge der Wörter, die mit einem a beginnen, dann kein weiteres a enthalten und mindestens ein b und ein c enthalten. Der folgende Zweiwegautomat überprüft zunächst, ob ein Wort ein führendes a enthält; dann sucht er ein b; wenn er das b gefunden hat, fährt er in der Eingabe wieder nach links zurück und sucht dann ein c und bewegt den Kopf dann in der Eingabe ganz nach rechts:



Zum besseren Verständnis geben wir die Bedeutung der verschiedenen Zustände an:

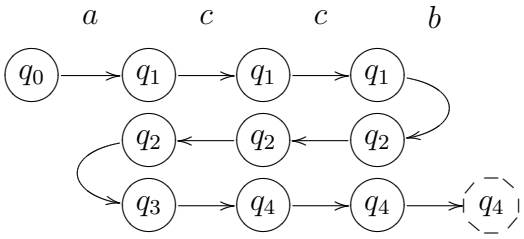
- q_0: überprüfen, ob das erste Zeichen ein a ist
- q_1: erstes b suchen
- q_2: b gefunden, zurück nach links zum Wortanfang
- q_3: erstes c suchen
- q_4: c gefunden, weiter nach rechts zum Wortende
- q_5: Fangzustand; wird erreicht, wenn das erste Zeichen kein a, oder ein späteres Zeichen ein a ist; in diesem Fall wird das Wort nicht akzeptiert.

Es gibt für diese Sprache auch einen (relativ) einfachen deterministischen Automaten. Einen Zweiwegautomaten zu benutzen, erscheint hier relativ künstlich (das mag sich bei einem größeren Alphabet ändern). Hier geht es uns nur um ein Beispiel – wenn auch ein etwas "blödes".

Wir geben nun eine *Berechnung* des Zweiwegautomaten für das Wort $accb \in L(A)$ an:

$$
\begin{array}{rcccc}
 & q_0 a & c & c & b \\
\vdash & a q_1 c & c & b \\
\vdash & a & c q_1 c & b \\
\vdash & a & c & c q_1 b \\
\vdash & a & c q_2 c & b \\
\vdash & a q_2 c & c & b \\
\vdash & q_2 a & c & c & b \\
\vdash & a q_4 c & c & b \\
\vdash & a & c q_4 c & b \\
\vdash & a & c & c q_4 b \\
\vdash & a & c & c & b q_4
\end{array}
$$

Etwas platzsparender ist die folgende Repräsentation der Berechnung des Zweiwegautomaten:



Achtung: In den ungeraden Zeilen beziehen sich die Zustände auf das Zeichen rechts vom Zustand; in den geraden Zeilen beziehen sich die Zustände auf das Zeichen links vom Zustand. Die Sequenz der untereinander stehenden Zustände nennen wir *Kreuzungsfolgen* (vgl. Abb. 2.17). Die Kreuzungsfolgen werden wir später benutzen, um die Äquivalenz zu endlichen Automaten zu beweisen.

Bemerkung: Offensichtlich ist jeder deterministische und vollständige Automat ein Spezialfall eines Zweiwegautomaten.
Frage: Kann man mit Zweiwegautomaten mehr Sprachen akzeptieren als mit endlichen Automaten?

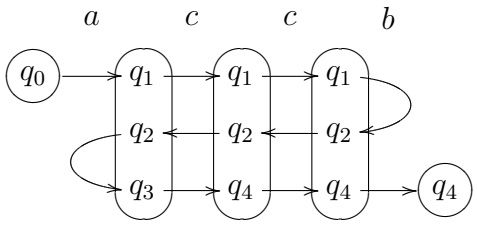


Abbildung 2.17. Die Kreuzungsfolgen einer Berechnung.

Die Antwort auf diese Frage ist nein. Aber warum?
Idee: Wir können die Kreuzungsfolgen als Zustände eines endlichen Automaten auffassen und dann die Übergänge zwischen ihnen charakterisieren.

> ***Fragen**: Wie viele Kreuzungsfolgen gibt es? Wann sind Übergänge zwischen zwei Kreuzungsfolgen möglich?*

Beobachtungen:

1. Keine Kreuzungsfolge eines akzeptierten Wortes besitzt zwei ungerade Positionen mit demselben Zustand oder zwei gerade Positionen mit demselben Zustand; denn sonst hat die entsprechende Berechnung einen Zyklus (vgl. Abb. 2.18).

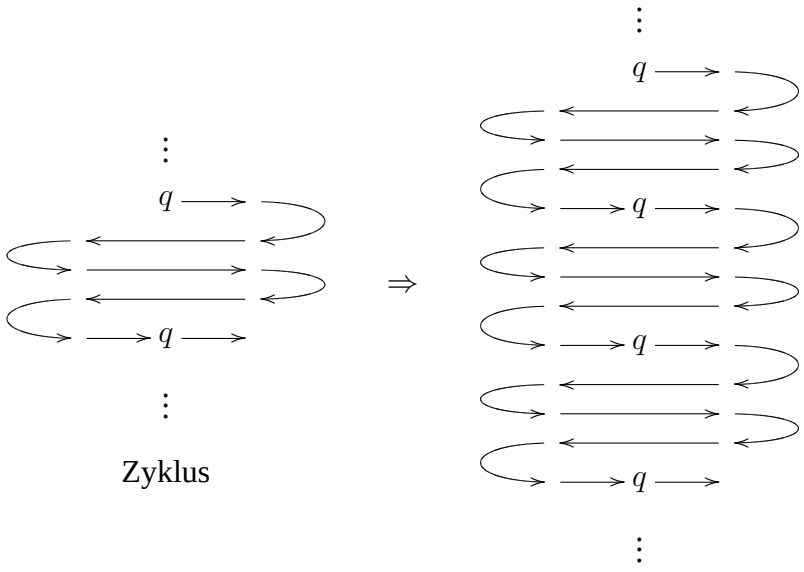


Abbildung 2.18. Derselbe Zustand an zwei gerade oder an zwei ungeraden Positionen: ein Zyklus.

   Damit können die Kreuzungsfolgen eines akzeptierten Wortes höchstens die Länge $2 \cdot |Q|$ haben. Wir müssen also nur endlich viele Kreuzungsfolgen betrachten.

2. Jede Kreuzungsfolge eines akzeptierten Wortes hat eine ungerade Länge.

3. Die erste Kreuzungsfolge *eines* beliebigen Wortes ist q_0.

4. Die letzte Kreuzungsfolge eines akzeptierten Wortes ist q für $q \in F$.

5. Insbesondere haben die erste und die letzte Kreuzungsfolge immer die Länge 1.

Jetzt müssen wir uns nur noch die möglichen Übergänge zwischen den Kreuzungsfolgen eines akzeptierten Wortes überlegen. Wir definieren diese Übergänge induktiv (und zwar für jedes $a \in \Sigma$ einzeln). Für diese induktive Definition benötigen wir jedoch auch die Übergänge zwischen Kreuzungsfolgen gerader Länge. Abbildung 2.19 gibt die induktive Definition der Übergänge zwischen den Kreuzungsfolgen.
Wir können also einen endlichen Automaten konstruieren, der dieselbe Sprache akzeptiert wie der Zweiwegautomat.

- Die Zustände des endlichen Automaten sind die "gültigen Kreuzungsfolgen".

- Der Startzustand ist die Kreuzungsfolge q_0.

- Die Endzustände sind die Kreuzungsfolgen q mit $q \in F$.

- Die Übergänge sind die induktiv definierten Übergänge zwischen den ungeraden Kreuzungsfolgen für jedes $a \in \Sigma$

Satz 2.17 *Die von einem Zweiwegautomaten akzeptierten Sprachen sind regulär.*

Beweis: Siehe Vorbemerkungen. □

Bemerkung: Auch für nicht-deterministische Zweiwegautomaten ist Satz 2.17 gültig. Der Beweis erfordert jedoch noch weitere Überlegungen und ist noch etwas aufwendiger.

5.2 Reguläre Grammatiken / rechtslineare Grammatiken

Nun charakterisieren wir die regulären Sprachen über Grammatiken einer speziellen Form: die sog. *regulären Grammatiken*.

> *Diese Charakterisierung ist viel wichtiger als der Zweiwegautomat. Wir stellen sie zuletzt vor, da die Charakterisierung der regulären Sprachen über Grammatiken einen guten Übergang zum nächsten Kapitel, den kontextfreien Sprachen ermöglicht.*

Definition 2.18 (Rechtslineare Grammatik) *Eine Grammatik $G = (V, \Sigma, P, S)$ heißt* rechtslinear*, wenn jede Produktion die Form*

$$A \to wB \qquad \textit{oder} \qquad A \to w$$

für $A, B \in V$ und $w \in \Sigma^*$ hat.

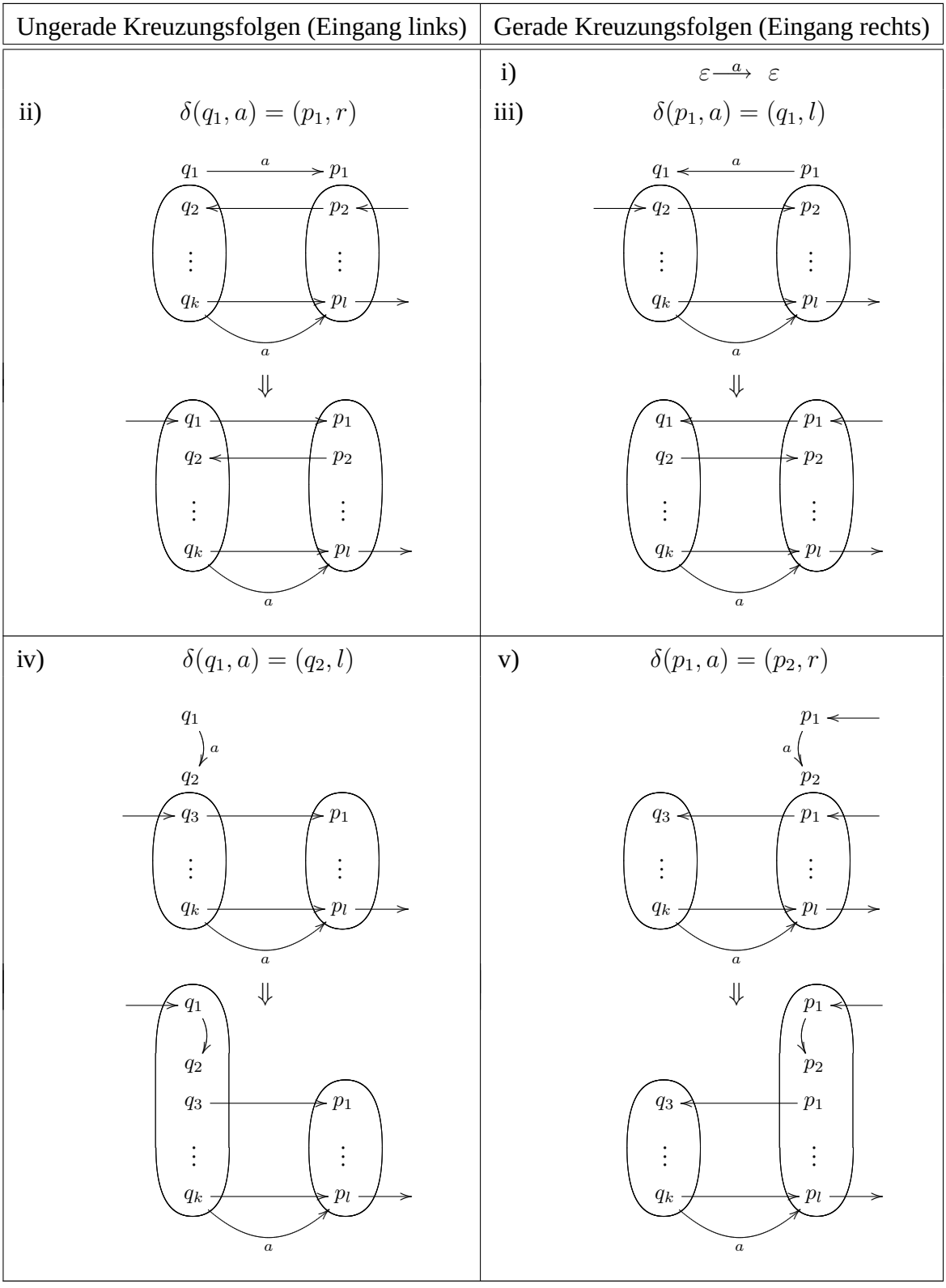


Abbildung 2.19. Induktive Definition der Übergänge zwischen den Kreuzungsfolgen.

Analog kann man linkslineare Grammatiken *mit Regeln der Form $A \to Bw$ und $A \to w$ definieren. Da die regulären Sprachen unter Spiegelung abgeschlossen sind, charakterisieren die linkslinearen Grammatiken die gleiche Sprachklasse wie die rechtslinearen Grammatiken, nämlich die regulären Sprachen. Deshalb faßt man rechts- und linkslineare Grammatiken unter dem Oberbegriff* reguläre Grammatik *zusammen. Sie werden auch Typ-3-Grammatiken genannt.*
Später werden wir auch noch die linearen Grammatiken *mit Regeln der Form $A \to wBv$ und $A \to w$ etwas genauer untersuchen. Allerdings lassen sich mit linearen Grammatiken wesentlich mehr Sprachen als die regulären Sprachen beschreiben (z.B. kann man die Sprache der Palindrome mit Hilfe einer linearen Grammatik beschreiben). Die von den linearen Grammatiken erzeugten Sprachen heißen* linear.

Satz 2.19 *Eine Sprache L ist genau dann regulär, wenn es eine rechtslineare Grammatik G mit $L = L(G)$ gibt.*

Beweis:
"⇐": Wir beweisen zuerst, dass es zu jeder rechtslineare Grammatik G einen endlichen Automaten gibt, der die entsprechende Sprache akzeptiert.
Idee:

$$S \Rightarrow_G w_1A_1 \Rightarrow_G w_1w_2A_2 \Rightarrow_G \ \ldots \ \Rightarrow_G w_1 \ldots w_nA_n \Rightarrow_G w_1 \ldots w_nw_{n+1}$$

$$S \xrightarrow{w_1} \ A_1 \xrightarrow{w_2} \ A_2 \longrightarrow \ldots \xrightarrow{w_n} \ A_n \xrightarrow{w_{n+1}} \ \bullet$$

Sei $G = (V, \Sigma, P, S)$, dann definieren wir den Automaten A mit $A = (V \cup \{\bullet\}, \Sigma, \delta, S, \{\bullet\})$.

$$\delta = \{(B, w, C) \mid B \to wC \in P\} \cup \{(B, w, \bullet) \mid B \to w \in P\}$$

Offensichtlich gilt:

- $(S, w, B) \in \delta^*$ gdw. wB ist Satzform von G.
- $(S, w, \bullet) \in \delta^*$ gdw. w ist Satz von G.

Beweis: Induktion über die Definition der Ableitung und der Definition von δ^*.

Damit gilt $L(A) = L(G)$ und somit ist $L(G)$ regulär.

„⇒": Nunmehr beweisen wir die Gegenrichtung, d.h. für eine reguläre Sprache L gibt es eine rechtslineare Grammatik G mit $L(G) = L$. Da L regulär ist, gibt es einen endlichen Automaten $A = (Q, \Sigma, \delta, q_0, F)$ mit $L(A) = L$. Wir definieren nun die Grammatik $G = (Q, \Sigma, P, q_0)$ mit:

$$P = \{q \to wq' \mid (q, w, q') \in \delta\} \cup \{q \to w \mid (q, w, q') \in \delta \wedge q' \in F\}$$

Wieder gilt:

- wq ist Satzform von G gdw. $(q_0, w, q) \in \delta^*$.
- w ist Satz von G gdw. $(q_0, w, q) \in \delta^*$ und $q \in F$.

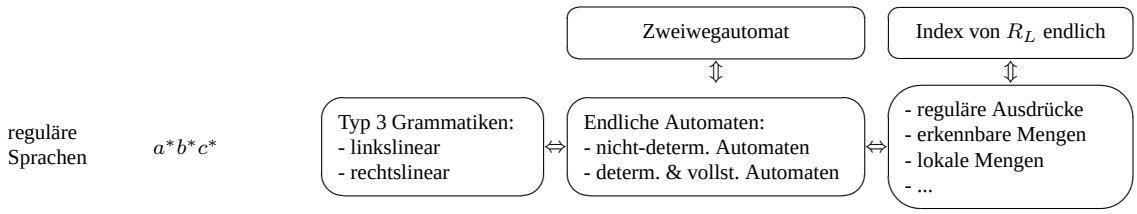
Also gilt $L = L(G)$; offensichtlich ist G rechtslinear. □

Bemerkungen: Automaten und rechtslineare Grammatiken lassen sich im wesentlichen 1 zu 1 ineinander übersetzen. Das gleiche selbstverständlich auch für linkslineare Grammatiken.

Redaktioneller Hinweis: Hier könnte man (irgendwann einmal) noch Abschnitte für erkennbare Mengen, lokale Mengen und Gleichungssysteme zur Charakterisierung regulärer Sprachen einführen.

5.3 Zusammenfassung

Unsere Tabelle aus Abschnitt 2 können wir nun wie folgt ergänzen:



6 Erweiterungen endlicher Automaten

In Abschnitt 1 haben wir endliche Automaten kennengelernt. Die dort vorgestellte Version geht im wesentlichen auf Rabin und Scott zurück. Sie werden daher auch *Rabin-Scott-Automaten* genannt.

Rabin-Scott-Automaten sind ein Konzept und keine mathematische Definition.

Es gibt aber noch viele weitere Konzepte für Automaten. Wir betrachten hier als Beispiel zwei verschiedene Versionen von *endlichen Automaten mit Ausgabe*: *Mealy*- und *Moore*-Automaten. Hier betrachten wir nur die deterministischen und vollständigen Versionen dieser Automaten. Die Definition nicht-deterministischer Automaten mit Ausgabe würde hier zu weit führen.

Es gibt noch viele weiter Konzepte von Automaten. Beispielsweise kann man mit Büchi-Automaten *unendlich lange Wörter akzeptieren. Dies spielen bei der Verifikation interaktiver bzw. reaktiver Systeme eine wichtige Rolle. Auch sog.* State-Charts *stellen eine Variante von Automaten dar, die man zur Modellierung reaktiver Systeme nutzt (vgl. Zustandsdiagramme in UML).*
Büchi-Automaten und State-Charts sind jedoch nicht Gegenstand dieser Vorlesung, sondern Gegenstand von Spezialvorlesungen.

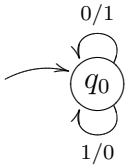
6.1 Automaten mit Ausgabe

Bei Automaten mit Ausgabe geht es nicht darum eine Sprache zu definieren (oder ein Wort einer Sprache zu akzeptieren). Vielmehr wird ein Eingabewort gelesen und daraus eine Ausgabewort erzeugt. Die Automaten mit Ausgabe berechnen also Funktionen.

Beispiel 2.13
Wir betrachten als Beispiel zwei Automaten, die das bitweises Komplement eines Bit-String ausgeben. Diese Automaten sind in Abb. 2.20 dargestellt.
Welche der Varianten besser, schöner oder eleganter ist, ist Geschmackssache und hängt auch davon ab, zu welchem Zweck der Automat benutzt wird.

Mealy-Automat

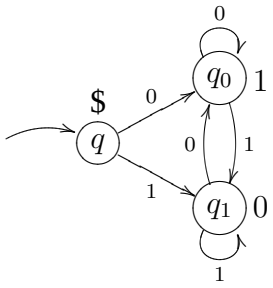
Der Mealy-Automat gibt bei jedem Zustandsübergang des Automaten ein Zeichen aus. An jeder Kante des Mealy-Automaten stehen deshalb zwei Zeichen, die durch einen Schrägstrich getrennt sind. Das Zeichen vor dem Schrägstrich ist das Zeichen, das beim Übergang gelesen wird, das Zeichen nach dem Schrägstrich ist das Zeichen, das beim Übergang ausgegeben.



Eingabe:	1	1	0
	$q_0 \xrightarrow{1/0} q_0$	$\xrightarrow{1/0} q_0$	$\xrightarrow{0/1} q_0$
Ausgabe:	0	0	1

Moore-Automat

Der Moore-Automat gibt in jedem Zustand (bzw. beim Erreichen eines Zustandes) ein Zeichen aus. Das auszugebende Zeichen steht neben dem entsprechenden Zustand. Das $-Symbol ist ein Sonderzeichen für den ersten Zustand.



Eingabe:	1	1	0	
	$q \longrightarrow$	$q_1 \longrightarrow$	$q_1 \longrightarrow$	q_0
Ausgabe:	$	0	0	1

Abbildung 2.20. Zwei verschiedene Varianten von Automaten mit Ausgabe.

6.1.1 Mealy-Automaten

Hier geben wir die formale Definition des Mealy-Automaten und der von ihm definierten Funktion an.

Definition 2.20 (Mealy-Automat) *Ein* Mealy-Automat A *ist definiert durch*

- *eine endliche Menge* Q *von* Zuständen*,*
- *zwei Alphabete* Σ *und* Δ*, wobei* Σ *das* Eingabe- *und* Δ *das* Ausgabe-Alphabet *ist,*
- *eine* Übergangsfunktion $\delta : Q \times \Sigma \to Q$*,*

  δ ist eine Funktion, da wir nur vollständige und deterministische Automaten betrachten.

- *einer* Ausgabefunktion $\lambda : Q \times \Sigma \to \Delta$ *und*
- *einem* Startzustand $q_0 \in Q$*.*

Wir schreiben dafür $A = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$*. Wir definieren* $\lambda^* : Q \times \Sigma^* \to \Delta^*$ *induktiv durch:*

- $\lambda^*(q, \varepsilon) = \varepsilon$
- $\lambda^*(q, aw) = \lambda(q, a) \circ \lambda^*(\delta(q, a), w)$

Für ein Wort $w \in \Sigma^*$ *nennen wir* $\lambda^*(q_0, w)$ *die* Antwort *von* A *auf die Eingabe* w*.*

6.1.2 Moore-Automat

Hier geben wir die formale Definition des Moore-Automaten und der von ihm definierten Funktion an. Die Definition ist ähnlich zu der des Mealy-Automaten; der Unterschied besteht lediglich in der Definition von λ und λ^*.

Definition 2.21 (Moore-Automat) *Ein* Moore-Automat A *ist definiert durch*

- *eine endlichen Menge* Q *von* Zuständen*,*
- *zwei Alphabete* Σ *und* Δ*, wobei* Σ *das* Eingabe- *und* Δ *das* Ausgabe-Alphabet *ist,*
- *eine* Übergangsfunktion $\delta : Q \times \Sigma \to Q$*,*
- *eine* Ausgabefunktion $\lambda : Q \to \Delta$ *und*

  Die Ausgabe ist beim Moore-Automaten im Gegensatz zum Mealy-Automaten nur vom Zustand abhängig.

- *einem* Startzustand $q_0 \in Q$

Wir schreiben dafür $A = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$*. Wir definieren* $\lambda^* : Q \times \Sigma^* \to \Delta^*$ *induktiv durch:*

- $\lambda^*(q, \varepsilon) = \lambda(q)$
- $\lambda^*(q, aw) = \lambda(q) \circ \lambda^*(\delta(q, a), w)$

Für ein Wort $w \in \Sigma^*$ *nennen wir* $\lambda^*(q_0, w)$ *die* Antwort *von* A *auf die Eingabe* w*.*

Bemerkung: Beim Moore-Automaten führt jede Eingabe (auch ε) auch zu einer Ausgabe $x \in \Delta^*$ mit $|x| \geq 1$! Trotzdem sind Mealy- und Moore-Automaten im wesentlichen äquivalent:

Satz 2.22

1. *Für jeden Moore-Automaten A gibt es einen Mealy-Automaten A' derart, daß für alle Wörter $w \in \Sigma$ die Antwort des Mealy-Automaten A' auf w das Wort $v \in \Delta^*$ ist, wenn $\$v$ die Antwort des Moore-Automaten A auf w ist.*

2. *Für jeden Mealy-Automaten A' gibt es einen Moore-Automaten A derart, daß für alle Wörter $w \in \Sigma$ die Antwort des Mealy-Automaten A' auf w das Wort $v \in \Delta^*$ ist, wenn $\$v$ die Antwort des Moore-Automaten A auf w ist.*

Beweis:
Idee: Die Überführung eines Moore-Automaten in einen entsprechenden Mealy-Automaten ist einfach. Das Zeichen, das im Moore-Automaten beim Erreichen des Zustandes ausgegeben wird, wird im Mealy-Automaten bereits beim Übergang in diesen Zustand ausgegeben.
Die Überführung eines Mealy-Automaten in einen entsprechenden Moore-Automaten ist etwas komplizierter, da bei Übergängen in denselben Zustand verschiedene Ausgaben möglich sind. Daher müssen die Zustände entsprechend repliziert werden (für jede mögliche Ausgabe eine eigene Version des Zustandes).

Ein ausführlicher Beweis wird in der Übung geführt (Blatt 7, Aufgabe 1). □

Bemerkung: Mealy- und Moore-Automaten sind nicht zum Erkennen von Sprachen gedacht, sondern zum Berechnen (von sehr einfachen) Funktionen. Deshalb kommen sie auf unserer "formal-sprachlichen Landkarte" nicht vor. Natürlich gibt es Analogien zu den endlichen Automaten (Rabin-Scott-Automaten). Aber es gibt auch deutliche Unterschiede. Beispielsweise unterscheidet sich die Ausdrucksmächtigkeit der deterministischen und der nicht-deterministische Automaten mit Ausgabe.

Kapitel 3

Kontextfreie Sprachen

1 Kontextfreie Grammatiken

Reguläre Sprachen stellen eine einfache Möglichkeit dar, Sprachen zu definieren und auch zu analysieren. Außerdem haben sie sehr schöne Eigenschaften: Sie sind unter vielen Operationen abgeschlossen, und die meisten Probleme sind für reguläre Sprachen entscheidbar.
Allerdings sind viele praktisch relevante Sprachen nicht regulär. Beispielsweise ist die Sprache der korrekten Klammerausdrücke nicht regulär. Generell lassen sich verschachtelte Strukturen (beliebiger Tiefe) nicht als reguläre Sprache formulieren, da man mit regulären Sprachen nicht (beliebig weit) zählen kann. Verschachtelte Strukturen kommen aber in vielen Bereichen der Informatik vor (z.B. in Ausdrücken oder bei ineinander verschachtelten Schleifen oder Alternativen).
In diesem Abschnitt betrachten wir nun eine Sprachklasse, die auf die Definition verschachtelter Strukturen zugeschnitten ist: die *kontextfreien Sprachen*.

1.1 Motivation und Definition

Ähnlich wie bei regulären Sprachen, gibt es verschiedene Charakterisierungen von kontextfreien Sprachen: eine über das Automatenmodell und eine über die Grammatik. Für kontextfreie Sprachen ist die einfachste Charakterisierung wohl die über *kontextfreie Grammatiken*.
Kontextfreie Grammatiken oder Varianten davon (z.B. die *Backus-Naur-Form*) sind DAS Mittel zur Beschreibung der Syntax von Programmiersprachen (oder eines wesentlichen Anteils davon).

Historisch wurden kontextfreie Grammatiken zunächst von einem Linguisten definiert (N. Chomsky).

Definition 3.1 (Kontextfreie Grammatik) *Eine Grammatik $G = (V, \Sigma, P, S)$ heißt* kontextfrei, *wenn gilt $P \subseteq V \times (V \cup \Sigma)^*$. Eine Sprache L heißt* kontextfrei, *wenn es eine kontextfreie Grammatik G gibt, die L erzeugt (d.h. $L = L(G)$).*

Jede Produktion einer kontextfreien Grammatik hat also die Form $A \to \alpha$ mit $A \in V$ und $\alpha \in (V \cup \Sigma)^*$.
Da $\alpha = \varepsilon$ nicht ausgeschlossen ist, können die Produktionen auch die Form $A \to \varepsilon$ haben.

Zur Erinnerung wiederholen wir hier nochmals kurz die wichtigsten Begriffe und Definitionen für Grammatiken (siehe Definition 1.2 auf Seite 9) für den Spezialfall der kontextfreien Grammatiken:

- Direkte Ableitbarkeit: $\alpha \Rightarrow_G \beta$ *genau dann, wenn* $A \to \beta' \in P$ *und* $\gamma, \gamma' \in (V \cup \Sigma)^*$ *mit* $\alpha = \gamma A \gamma'$ *und* $\beta = \gamma \beta' \gamma'$ *existieren.*
- Ableitbarkeit: $\alpha \Rightarrow_G^* \beta$
- Ableitbarkeit in i Schritten: $\alpha \Rightarrow_G^i \beta$
- Satzform: $\alpha \in (V \cup \Sigma)^*$ *mit* $S \Rightarrow_G^* \alpha$.
- Satz: $w \in \Sigma^*$ *mit* $S \Rightarrow_G^* w$.

Die Bezeichnung „kontextfrei“ beruht auf der Tatsache, daß bei Anwendung einer Produktion $A \to \beta$ die Variable A unabhängig von dem Kontext (γ und γ'), in dem sie vorkommt, durch β ersetzt wird. Die Anwendung von Produktionen an verschiedenen Stellen einer Satzform sind also vollkommen unabhängig voneinander. Dies wird in der Aussage 3 des folgenden Lemmas formalisiert; als weitere Konsequenz ergibt sich, daß sich jede Ableitung als Baum darstellen läßt. Darauf gehen wir in Abschnitt 1.2 noch genauer ein.

Lemma 3.2 *Sei* $G = (V, \Sigma, P, S)$ *eine Grammatik. Es gilt:*

1. *Seien* $\alpha, \beta \in (V \cup \Sigma)^*$ *mit* $\alpha \Rightarrow_G^i \beta$. *Dann gilt für alle* $\gamma, \gamma' \in (V \cup \Sigma)^*$ *auch* $\gamma\alpha\gamma' \Rightarrow_G^i \gamma\beta\gamma'$.

2. *Für* $l = 1, \ldots, n$ *seien* $\alpha_l, \beta_l \in (V \cup \Sigma)^*$ *mit* $\alpha_l \Rightarrow_G^{i_l} \beta_l$ *und* $i_l \in \mathbb{N}$. *Dann gilt* $\alpha_1 \ldots \alpha_n \Rightarrow_G^i \beta_1 \ldots \beta_n$ *mit* $i = \sum_{l=1}^n i_l$.

3. *Sei nun* G *kontextfrei. Für jedes* $l = 1, \ldots, n$ *sei* $\alpha_l \in (V \cup \Sigma)^*$ *und sei* $\beta \in (V \cup \Sigma)^*$, *so daß gilt* $\alpha_1 \ldots \alpha_n \Rightarrow_G^i \beta$. *Dann existiert für jedes* $l = 1, \ldots, n$ *ein* $\beta_l \in (V \cup \Sigma)^*$ *und ein* $i_l \in \mathbb{N}$ *mit* $\alpha_l \Rightarrow_G^{i_l} \beta_l$, $\beta = \beta_1 \ldots \beta_n$ *und* $i = \sum_{l=1}^n i_l$.

Redaktioneller Hinweis: Zu Aussage 3 könnte man noch eine graphische Darstellung machen.

Die Aussagen 1 und 2 von Lemma 3.2 gelten für jede Grammatik; lediglich für die „Umkehrung“, also Aussage 3, müssen wir voraussetzen, daß G kontextfrei ist.

Beweis: Das Lemma wird in der Übung (Blatt 7, Aufgabe 3) bewiesen. Dazu wenden wir im wesentlichen die Definition der direkten Ableitbarkeit an und weisen die Aussagen durch Induktion über die Länge der Ableitung nach. □

1.2 Ableitungsbäume

Eine weitere Konsequenz der Einschränkung auf kontextfreie Grammatiken ist, daß sich jede Ableitung als Baum darstellen läßt. Diese Darstellung ist in vielen Fällen zweckmäßiger als die Ableitung selbst; denn der Baum liefert die syntaktische Struktur des Wortes. Dieser Baum wird *Ableitungsbaum* oder *Syntaxbaum* genannt.

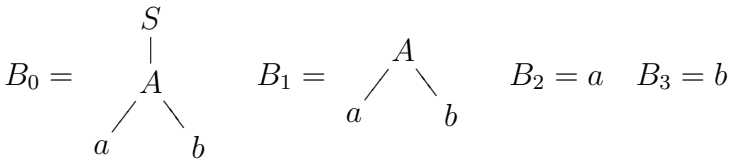
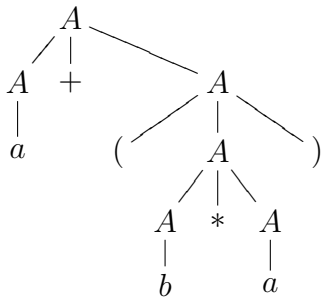


Abbildung 3.1. Ein Baum B_0 mit seinen Teilbäumen B_1, B_2 und B_3.

Beispiel 3.1
Wir betrachten nochmals die Grammatik, die wir bereits aus dem Beispiel 1.1 auf Seite 9 kennen. Wir geben den Ableitungsbaum für das Wort $a + (b * a)$ an:

$$
\begin{array}{lcl}
A & \rightarrow & A + A \\
A & \rightarrow & A * A \\
A & \rightarrow & (A) \\
A & \rightarrow & a \\
A & \rightarrow & b
\end{array}
$$



An der Wurzel[1] des Ableitungsbaumes steht das Axiom. Jede Verzweigung an einem Knoten des Baumes entspricht einer Produktion der Grammatik. Die Blätter des Baumes sind die Zeichen des Terminalalphabetes. Formal definieren wir die Ableitungsbäume induktiv von unten nach oben. Da wir die Bäume von unten nach oben (d.h. von den Blättern zur Wurzel) aufbauen, müssen wir in der induktiven Definition auch Bäume betrachten, die bei einer beliebigen Variable beginnen (vgl. Abb. 3.1). Um die Definition zu vereinfachen, betrachten wir sogar ε und alle Zeichen a aus dem Terminalalphabet als Bäume mit genau einem Knoten. Wir definieren also etwas allgemeiner x-Bäume, deren Wurzel ein Symbol x ist, wobei $x \in V \cup \Sigma \cup \{\varepsilon\}$. In der induktiven Definition der Bäume, ordnen wir jedem Baum außerdem ein Wort zu, das wir den *Rand* des Baumes nennen und dem dem abgeleiteten Wort entspricht.

> *Achtung: Hier weichen wir kurzzeitig von unseren Konventionen ab. Der Bezeichner x steht hier nicht für ein Wort, sondern für ein Zeichen aus V oder Σ oder für das leere Wort ε.*

Definition 3.3 (Ableitungsbaum, x-Baum, Rand)
Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. Für $x \in V \cup \Sigma \cup \{\varepsilon\}$ definieren wir induktiv die Menge der x-Bäume *und ihren* Rand *wie folgt:*

1. *Für $x \in \Sigma \cup \{\varepsilon\}$ ist x ein x-Baum und es gilt $Rand(x) = x$.*

[1] In der Informatik ist die Wurzel im Gegensatz zur Biologie oben. Die Bäume wachsen nach unten – und demzufolge wachsen die Bäume in der Informatik nicht in den Himmel.

2. *Wenn $A \to x_1 \dots x_n \in P$ eine Produktion mit $n \geq 1$ ist mit $x_i \in V \cup \Sigma$ für jedes $i \in \{1, \dots, n\}$, und falls B_i für jedes $i \in \{1, \dots, n\}$ ein x_i-Baum mit Rand$(x_i) = w_i$ ist, dann ist*

$$B = \quad \begin{array}{ccccc} & & A & & \\ & \swarrow & & \searrow & \\ B_1 & & \cdots & & B_n \end{array}$$

ein A-Baum mit $Rand(B) = w_1 \dots w_n$.

Achtung: Auf den Kindern eines Knotens ist eine Ordnung definiert; Ableitungsbäume sind also geordnete Bäume.

3. *Für eine Produktion $A \to \varepsilon \in P$ ist*

$$B = \quad \begin{array}{c} A \\ | \\ \varepsilon \end{array}$$

ein A-Baum mit $Rand(B) = \varepsilon$.

Jeder S-Baum B heißt Ableitungsbaum *von G für das Wort Rand(B).*

Offensichtlich gibt es einen engen Zusammenhang zwischen den Ableitungsbäumen und den Ableitungen einer Grammatik:

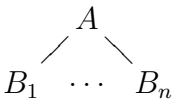
Satz 3.4 (Ableitungsbäume und Ableitungen)
Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik und $w \in \Sigma^*$. Dann gilt $S \Rightarrow_G^* w$ genau dann, wenn ein Ableitungsbaum B mit $Rand(B) = w$ existiert.

Beweis:
„⇒“: Zunächst zeigen wir, dass zu jeder Ableitung einer kontextfreien Grammatik G ein entsprechender Ableitungsbaum B existiert. Wir zeigen etwas allgemeiner, daß es für jedes $x \in V \cup \Sigma \cup \{\varepsilon\}$ und jede Ableitung $x \Rightarrow_G^* w$ einen x-Baum mit $Rand(B) = w$ gibt.
Wir konstruieren diesen Baum B induktiv über die Länge der Ableitung $x \Rightarrow_G^i w$:

$i = 0$: Aus $x \Rightarrow_G^0 w$ folgt $x = w$; insbesondere gilt dann wegen $w \in \Sigma^*$ auch $x \in \Sigma \cup \{\varepsilon\}$. Gemäß Definition 3.3.1 ist x ein x-Baum mit $Rand(x) = x = w$.

$i \to i + 1$: Gelte also $x \Rightarrow_G^{i+1} w$. Gemäß Definition 1.2 gibt es eine Produktion $A \to x_1 \dots x_n \in P$ mit $A = x$, $x_l \in V \cup \Sigma$ und $x_1 \dots x_n \Rightarrow_G^i w$.
Wir unterscheiden nun zwei Fälle:

$n \geq 1$**:** Gemäß Lemma 3.2.3 gibt es dann Wörter $w_1, \ldots, w_n$ und Zahlen $i_1, \ldots, i_n \in \mathbb{N}$ mit $x_l \Rightarrow_G^{i_l} w_l$, $i_l \leq i$ und $w = w_1 \ldots w_n$.
Gemäß der Induktionsvoraussetzung gibt es dann für jedes $l \in \{1, \ldots, n\}$ einen x_l-Baum B_l mit $Rand(B_l) = w_l$. Gemäß Definition 3.3.2 ist dann

$B =$ 

ein A-Baum mit $Rand(B) = w_1 \ldots w_n = w$.

$n = 0$**:** In diesem Fall gilt $w = \varepsilon$ und die Produktion ist $A \to \varepsilon$. Gemäß Definition 3.3.3 ist dann

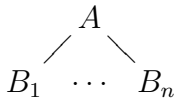
$B =$ 

ein A-Baum mit $Rand(B) = \varepsilon$.

„⇐“: Nun zeigen wir, daß zu jedem Ableitungsbaum B einer kontextfreien Grammatik G eine entsprechende Ableitung existiert. Wir konstruieren dazu induktiv über den Aufbau eines x-Baumes B eine Ableitung $x \Rightarrow_G^* w$ mit $Rand(B) = w$.

1. Sei $B = x \in \Sigma \cup \{\varepsilon\}$ ein x-Baum mit $Rand(B) = x$. Wegen der Reflexivität der Ableitbarkeit gilt offensichtlich $x \Rightarrow_G^* x$.

2. Für eine Produktion $A \to x_1 \ldots x_n$ mit $n \geq 1$ und x_l-Bäume B_l mit $Rand(B_l) = w_l$ sei

   $B =$ 

   ein A-Baum mit $Rand(B) = w_1 \ldots w_n = w$. Gemäß Induktionsvoraussetzung gibt es für jedes $l \in \{1, \ldots n\}$ eine Ableitung $x_l \Rightarrow_G^* w_l$. Mit Lemma 3.2.2 gilt somit $A \Rightarrow_G x_1 \ldots x_n \Rightarrow_G^* w_1 \ldots w_n$. Damit gilt $A \Rightarrow_G^* w$.

3. Für eine Produktion$A \to \varepsilon \in P$ sei

   $B =$ 

   ein A-Baum mit $Rand(B) = \varepsilon$. Gemäß Definition 1.2 gibt es dann eine Ableitung $A \Rightarrow_G \varepsilon$; also gilt insbesondere $A \Rightarrow_G^* \varepsilon$.

□

1.3 Eindeutigkeit und Mehrdeutigkeit

Im allgemeinen gibt es für ein Wort der erzeugten Sprache einer kontextfreien Sprache viele verschiedene Ableitungen, da die Ableitungsschritte an verschiedenen Stellen einer Satzform

unabhängig voneinander sind (vgl. Lemma 3.2) und deshalb in beliebiger Reihenfolge angewendet werden können.
Ableitungsbäume haben nun den Vorteil, daß sie die Reihenfolge von Ableitungsschritte an verschiedenen Stellen nicht festlegen. Es besteht also eine gewisse Chance, daß der Ableitungsbaum für ein Wort der erzeugten Sprache (bis auf Isomorphie) eindeutig ist.

Frage: Gibt es für jedes erzeugte Wort einer Grammatik einen eindeutigen Ableitungsbaum? Die Antwort ist Nein! In Beispiel 3.2 diskutieren wir eine mehrdeutige (nicht eindeutige) Grammatik. Tatsächlich ist es sehr einfach (vgl. Übung 7) mehrdeutige Grammatiken zu konstruieren. Das folgende Beispiel zeigt aber noch mehr: Mehrdeutigkeit ist in den meisten Fällen unerwünscht (z.B. bei der Syntaxanalyse bzw. im Compilerbau).

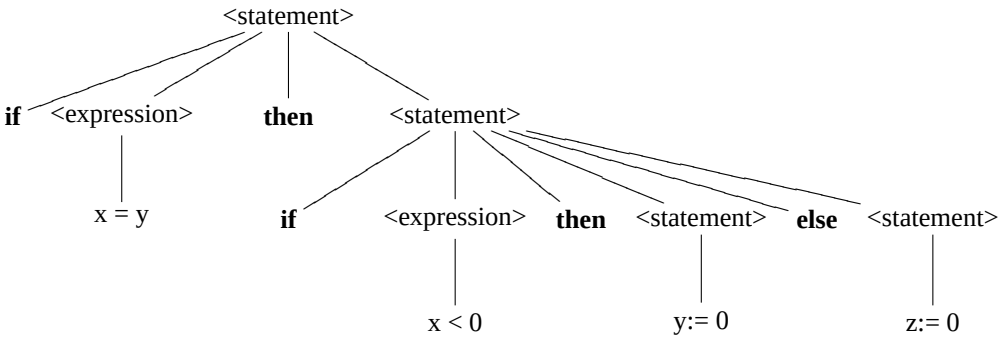
Beispiel 3.2 (Mehrdeutige Grammatik)
Wir betrachten eine Grammatik (bzw. einen Ausschnitt daraus), die die Syntax von „kurzen“ und „langen“ bedingten Anweisungen beschreibt, wobei wir die Regeln für elementare Anweisungen oder weitere Kontrollkonstrukte weglassen:

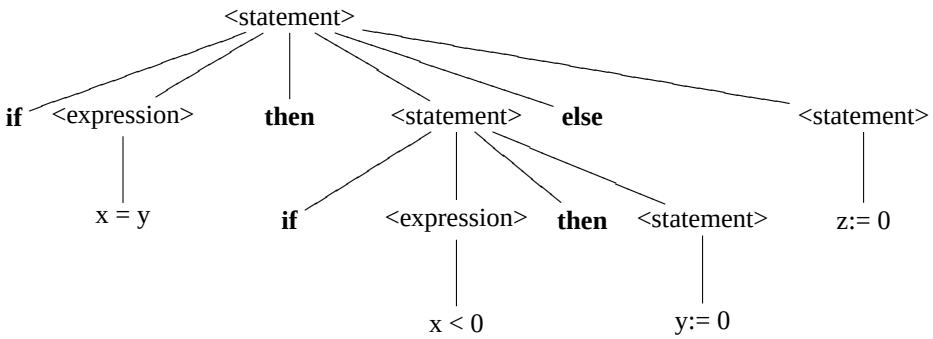
<statement> → **if** <expression> **then** <statement>
<statement> → **if** <expression> **then** <statement> **else** <statement>
<statement> → ...
<expression> → ...

Nun betrachten wir das Programm (das Wort)

if x = y **then if** x < 0 **then** y:= 0 **else** z:= 0

Für dieses Programm gibt es zwei verschiedene Ableitungsbäume:





Im ersten Fall gehört der **else**-Teil zur inneren **if**-Anweisung; im zweiten Fall gehört der **else**-Teil zur äußeren **if**-Anweisung. In konkreten Programmiersprachen dürfen solche Mehrdeutigkeiten nicht vorkommen, da die verschiedenen Ableitungsbäume verschiedene Bedeutung haben und somit zu einer verschiedenen Übersetzung des Programmes führen.

Man kann natürlich die Mehrdeutigkeit auflösen, indem man bei mehreren möglichen Ableitungsbäumen einen bestimmten auszeichnet. Dies ist aber unschön. Besser ist es, die Eindeutigkeit über die Grammatik zu gewährleisten.

In den allermeisten Fällen wünscht man sich, dass eine Grammatik *eindeutig* ist, d.h. daß es für jedes von der Grammatik erzeugte Wort einen eindeutigen Ableitungsbaum gibt:

Definition 3.5 (Mehrdeutigkeit) *Eine kontextfreie Grammatik heißt* mehrdeutig, *wenn es für mindestens ein Wort der erzeugten Sprache zwei verschiedene Ableitungsbäume gibt; anderenfalls heißt die Grammatik* eindeutig. *Eine kontextfreie Sprache heißt* inhärent mehrdeutig, *wenn jede kontextfreie Grammatik, die diese Sprache erzeugt, mehrdeutig ist (d.h. wenn es gibt keine eindeutige Grammatik gibt, die die Sprache erzeugt).*

Beispiel 3.3

1. Beispiele für mehrdeutige Grammatiken:

   - Beispiel 3.2.
   - $S \to \varepsilon \qquad S \to \mathsf{a}S \qquad S \to \mathsf{aa}S$
   - Es ist einfach, für eine kontextfreie Grammatik eine äquivalente mehrdeutige Grammatik anzugeben.

     Es gibt nur wenige kontextfreie Grammatiken, für die das nicht geht. Können Sie diese Grammatiken charakterisieren?

2. Ein Beispiel für eine inhärent mehrdeutige kontextfreie Sprache: Wir definieren zunächst $L_1 = \{a^n b^n c^k \mid n, k \in \mathbb{N}\}$ und $L_2 = \{a^k b^n c^n \mid n, k \in \mathbb{N}\}$. Dann ist die Sprache $L = L_1 \cup L_2$ inhärent mehrdeutig.

   Der Beweis, daß L inhärent mehrdeutig ist, ist relativ aufwendig. Deshalb geben wir hier nur eine Plausibilitätserklärung:

   - Für Wörter aus L_1 gibt es einen Ableitungsbaum aus G_1.
   - Für Wörter aus L_2 gibt es einen Ableitungsbaum aus G_2.
   - Es gibt aber auch Wörter $w \in L_1 \cap L_2$, für die es nun zwei Ableitungsbäume gibt, einen in G_1 und einen in G_2.

   Achtung: Diese Plausibilitätserklärung sollte nicht mit einem Beweis verwechselt werden. Das ist sie nicht. Das sieht man schon daran, daß nicht jede Vereinigung zweier kontextfreier Sprachen inhärent mehrdeutig ist.

Mit Definition 3.5 haben wir die Mehrdeutigkeit präzise definiert. Die Frage ist nun, ob wir einer Grammatik „ansehen“ können, ob sie mehrdeutig ist – und ggf. in eine äquivalente eindeutige Grammatik überführen können. Leider ist dies nicht möglich:

- Für eine kontextfreie Grammatik ist nicht entscheidbar, ob sie mehrdeutig ist oder nicht!

  Das können wir hier noch nicht beweisen.

- Selbst wenn wir wissen, daß eine Grammatik mehrdeutig ist, können wir dazu nicht immer eine äquivalente eindeutige Grammatik angeben! Im Beispiel 3.3.2 haben wir eine inhärent mehrdeutige kontextfreie Sprache angegeben; für die kann per Definition keine eindeutige kontextfreie Grammatik existieren.

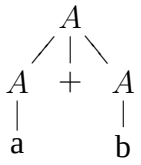
Ganz so dramatisch sind die negativen Resultate in der Praxis jedoch nicht:

- Für die meisten praktisch relevanten Sprachen gibt es eine eindeutige kontextfreie Grammatik. Dies trifft insbesondere auf die bedingten Anweisungen aus Beispiel 3.2 zu. Allerdings sind diese eindeutigen Grammatiken meist deutlich komplizierter.

  Sie können sich ja mal die LALR(1)-Grammatik der Java-Syntax in „The Java Language Specification“ (Chapter 19) ansehen.

- Für die Syntaxanalyse schränkt man aus Effizienzgründen die Grammatiken ohnehin noch viel stärker ein; diese stärkeren Bedingungen sind entscheidbar und implizieren die Eindeutigkeit der Grammatik (siehe Abschnitt 5).

Die Eindeutigkeit bzw. die Mehrdeutigkeit einer Grammatik wurde über die Ableitungsbäume definiert. Eingangs hatten wir bereits festgestellt, daß eine Definition direkt über die Ableitungen weniger sinnvoll ist, da es für einen Ableitungsbaum wegen der Vertauschbarkeit unabhängiger Ableitungsschritte im allgemeinen mehrere verschiedene korrespondierende Ableitungen gibt. Ein Beispiel dafür ist in Abb. 3.2 angegeben. Wenn jedoch die unabhängigen Ableitungs-



$$A \to A + A \to \mathrm{a} + A \to \mathrm{a} + \mathrm{b}$$

$$A \to A + A \to A + \mathrm{b} \to \mathrm{a} + \mathrm{b}$$

Abbildung 3.2. Ein Ableitungsbaum mit zwei verschiedenen korrespondierenden Ableitungen

schritte in einer Ableitung immer in einer festgelegten Reihenfolge angewendet werden, gibt es immer eine ausgezeichnete Ableitung. Dann läßt sich die Mehrdeutigkeit über diese kanonischen Ableitungen definieren.
Im wesentlichen gibt es zwei Möglichkeiten, diese kanonischen Ableitungen zu definieren. Entweder wird in jedem Ableitungsschritt eine Regel auf die am weitesten links stehende Variable angewendet (dann sprechen wir von einer *Linksableitung*), oder es wird in jedem Ableitungsschritt eine Regel auf die am weitesten rechts stehende Variable angewendet (dann sprechen wir von einer *Rechtsableitung*).

Redaktioneller Hinweis: Die folgende Defintion und der folgende Satz haben noch keine Nummer, damit die Numerierung zum handschriftlichen Skript paßt; in einer zukünfigen Version, werden auch diese Definition und dieser Satz eine Nummer erhalten.
Irgendwann könnte man bereits hier auch die Links- bzw. Rechtssatzformen definieren.

Definition (Links- und Rechtsableitung)
Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik und $\alpha, \beta \in (V \cup \Sigma)^*$.
Wir sagen β ist aus α direkt *linksableitbar*, wenn eine Produktion $A \to \beta' \in P$, ein $w \in \Sigma^*$ und ein $\gamma' \in (V \cup \Sigma)^*$ mit $\alpha = wA\gamma'$ und $\beta = w\beta'\gamma'$ existieren. Wir schreiben dann $\alpha \overset{L}{\Rightarrow}_G \beta$.
Wir sagen β ist aus α direkt *rechtsableitbar*, wenn eine Produktion $A \to \beta' \in P$, ein $w' \in \Sigma^*$ und ein $\gamma \in (V \cup \Sigma)^*$ mit $\alpha = \gamma Aw'$ und $\beta = \gamma\beta'w'$ existieren. Wir schreiben dann $\alpha \overset{R}{\Rightarrow}_G \beta$.

Der Unterschied zur allgemeimenen Ableitbarkeit besteht darin, daß der linke Kontext w bzw. der rechte Kontext w' hier ein Wort des Terminalalphabets sein muß.

Satz
Eine Grammatik ist genau dann mehrdeutig, wenn es ein Wort der erzeugten Sprache mit zwei verschiedenen Linksableitungen gibt.

Der Satz gilt natürlich ganz analog für Rechtsableitungen.

Beweis: Der Beweis baut auf den Beweisen von Satz 3.4 und Lemma 3.2 auf. Man muß zusätzlich zeigen, daß man aus zwei verschiedene Linksableitungen auch zwei verschiedenen Bäume konstruieren kann, und aus zwei verschiedenen Bäumen zwei verschiedene Linksableitungen. □

1.4 Normalformen

In diesem Abschnitt betrachten wir Spezialfälle kontextfreier Grammatiken: *Normalformen*. Wir werden sehen, daß es für jede kontextfreie Grammatik eine kontextfreie Grammatik in einer bestimmten Normalform gibt, bzw. daß man jede kontextfreie Grammatik effektiv in eine *äquivalente* Grammatik in dieser Normalform überführen kann. Zunächst definieren wir daher die *Äquivalenz* von Grammatiken.

Definition 3.6 (Äquivalenz von Grammatiken) *Zwei Grammatiken G_1 und G_2 heißen* äquivalent, *wenn sie dieselbe Sprache erzeugen (d.h. wenn gilt $L(G_1) = L(G_2)$).*

Achtung: In der Definition der Normalform verlangt man üblicherweise, daß es eine eindeutige Normalform für jedes betrachtete Objekt (in unserem Falle also für jede kontextfreie Grammatik) gibt. Dies ist bei unseren Normalformen nicht der Fall. Der Begriff Normalform hat sich hier aber fest etabliert; deshalb bleiben wir bei dem Begriff Normalform, auch wenn er etwas irreführend ist.

1.4.1 Vereinfachungen

Zunächst zeigen wir, daß man jede Grammatik auf eine bestimmte Weise vereinfachen kann: Wir eliminieren *nutzlose* Variablen, und Produktionen der Form $A \to \varepsilon$ (*ε-Produktionen*) und Produktionen der Form $A \to B$ (*Kettenproduktionen*).
Wir nennen eine Variable *nützlich*, wenn sie in wenigstens einer Ableitung eines Wortes vorkommt. Die Nützlichkeit zerfällt dabei in zwei Teilaspekte. Die *Produktivität* besagt, daß aus der Variablen wenigstens ein Wort abgeleitet werden kann; die *Erreichbarkeit* besagt, daß die Variable in einer (aus dem Axiom ableitbaren) Satzform auch vorkommt. Formal definieren wir:

Definition 3.7 (Nützliche, produktive und erreichbare Variablen)
Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik: Eine Variable $A \in V$ heißt

1. nützlich, *wenn ein Wort $w \in \Sigma^*$ und Kontexte $\gamma, \gamma' \in (V \cup \Sigma)^*$ mit $S \Rightarrow_G^* \gamma A \gamma' \Rightarrow_G^* w$ existieren. Andernfalls heißt A* nutzlos. *A heißt*

2. produktiv, *wenn ein Wort $w \in \Sigma^*$ mit $A \Rightarrow_G^* w$ existiert; A heißt*

3. erreichbar, *wenn Kontexte $\gamma, \gamma' \in (V \cup \Sigma)^*$ mit $S \Rightarrow_G^* \gamma A \gamma'$ existieren.*

Beobachtungen:

1. Wenn A nützlich ist, dann ist A auch produktiv und erreichbar.

   Dies folgt unmittelbar aus der Definition der Nützlichkeit, der Erreichbarkeit und der Produktivität einer Variablen (unter Ausnutzung von Lemma 3.2.3).

2. Wenn A produktiv und erreichbar ist, heißt das jedoch nicht unbedingt, daß A auch nützlich ist. Dazu betrachten wir das folgende Beispiel (bzw. Gegenbeispiel):

Beispiel 3.4
Beispiel einer Grammatik mit einer produktiven und nützlichen Variablen, die nicht nützlich ist:

$$\begin{array}{lcll} S & \to & AX & \rightsquigarrow A \text{ ist erreichbar} \\ A & \to & a & \rightsquigarrow A \text{ ist produktiv} \end{array}$$

Die Variable A ist nicht nützlich, da kein Satz ableitbar ist (da X nicht produktiv ist). Die einzige Ableitung dieser Grammatik ist $S \Rightarrow AX \Rightarrow aX$, wobei aX kein Satz ist.

Allerdings gilt die Aussage, wenn wir sie für alle Variablen einer Grammatik formulieren:

Lemma 3.8 *Wenn alle Variablen einer Grammatik produktiv und erreichbar sind, dann sind alle Variablen auch nützlich.*

Beweis: Wir nehmen an, daß alle Variablen der Grammatik produktiv und erreichbar sind. Wir müssen nun für jede Variable $A \in V$ zeigen, daß sie auch nützlich ist:
Gemäß Annahme ist A erreichbar, d.h. $S \Rightarrow_G^* \gamma A \gamma'$. Gemäß Annahme sind alle Variablen, also insbesondere die in γ und γ' und A selbst, produktiv; also existiert ein $w \in \Sigma^*$ mit $\gamma A \gamma' \Rightarrow_G^* w$ (mit Lemma 3.2.2). Die Variable A kommt also in einer Ableitung des Wortes w vor und ist somit nützlich. □

Beobachtungen:

1. Die Menge der produktiven Variablen aus V können wir iterativ wie folgt bestimmen:

$$V_0 = \{A \in V \mid A \to w \in P, w \in \Sigma^*\}$$
$$V_{i+1} = \{A \in V \mid A \to \alpha \in P, \alpha \in (\Sigma \cup V_i)^*\}$$

Offensichtlich gilt $V_0 \subseteq V_1 \subseteq V_3 \ldots$. Da $V_i \subseteq V$ für jedes $i \in \mathbb{N}$ endlich ist, gibt es ein $i \in \mathbb{N}$ mit $V_i = V_{i+1} = V_{i+2} = \ldots$. Diese Menge bezeichnen wir mit V^G_{prod}; sie ist effektiv bestimmbar und entspricht genau der Menge der produktiven Variablen der Grammatik.

Daß V^G_{prod} genau die Menge der nützlichen Variablen ist, werden wir in der Übung (Blatt 8, Aufgabe 1.b) noch etwas ausführlicher beweisen.

2. Die Menge der erreichbaren Variablen einer Grammatik können wir iterativ wie folgt bestimmen:

$$V_0 = \{S\}$$
$$V_{i+1} = V_i \cup \{A \in V \mid B \to \alpha A \beta \in P, B \in V_i\}$$

Wieder gibt es ein $i \in \mathbb{N}$ mit $V_i = V_{i+1} = V_{i+2} = \ldots$. Diese Menge bezeichnen wir mit V^G_{err}; sie ist effektiv bestimmbar und entspricht genau der Menge der erreichbaren Variablen der Grammatik.

Satz 3.9 (Entfernung nutzloser Variablen) *Für jede kontextfreie Grammatik G mit $L(G) \neq \varnothing$ gibt es effektiv eine äquivalente Grammatik G' ohne nutzlose Variablen.*

Beweis: Sei G eine kontextfreie Grammatik mit $G = (V, \Sigma, P, S)$. Wir konstruieren nun in zwei Schritten aus G eine äquivalente Grammatik G'' ohne nutzlose Variablen:

Schritt 1: Entferne alle nicht-produktiven Variablen (nebst den Produktionen, in denen sie vorkommen): $G' = (V^G_{prod}, \Sigma, P', S)$ mit $S \in V^G_{prod}$ (denn $L(G) \neq \varnothing$!) und mit $P' = P \cap (V^G_{prod} \times (V^G_{prod} \cup \Sigma)^*)$.

Das Entfernen von nicht-produktiven Variablen und den zugehörigen Produktionen verändert die erzeugte Sprache nicht. Also gilt $L(G') = L(G)$.

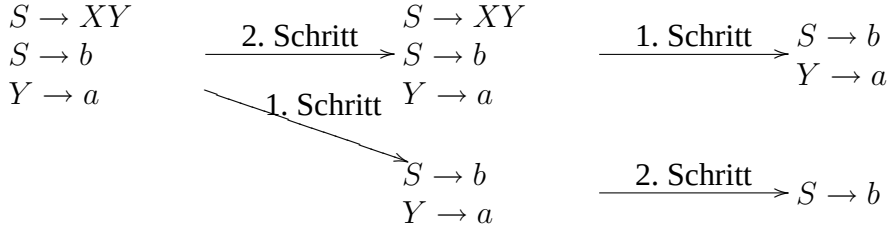
Schritt 2: Entferne nun alle nicht-erreichbaren Variablen aus G' (nebst den Produktionen, in denen sie vorkommen): $G'' = (V^{G'}_{err}, \Sigma, P'', S)$ mit $S \in V^{G'}_{err}$ (per Definition von $V^{G'}_{err}$) und mit $P'' = P' \cap (V^{G'}_{err} \times (V^{G'}_{err} \cup \Sigma)^*)$.

Das Entfernen von nicht-erreichbaren Variablen und den zugehörigen Produktionen verändert die erzeugte Sprache nicht. Also gilt $L(G'') = L(G') = L(G)$.

Jetzt ist jede Variable von G'' erreichbar (per Konstruktion von G'') und produktiv (erfordert etwas Überlegung). Nach Lemma 3.8 ist jede Variable von G'' nützlich; außerdem gilt $L(G'') = L(G)$. □

Bemerkungen:

1. Das Vertauschen der Schritte 1 und 2 im Beweis von Satz 3.9 führt nicht (immer) zum gewünschten Ergebnis:



Wenn Schritt 2 vor Schritt 1 ausgeführt wird, kann es nicht-erreichbare Variablen in der konstruierten Grammatik geben (im Beispiel ist dies die Variable Y).

2. Die Voraussetzung $L(G) = \varnothing$ ist notwendig, denn für $L(G) = \varnothing$ muß die Grammatik eine unproduktive Variable besitzen: Das Axiom S.

Diese Beobachtung liefert uns ein Entscheidungsverfahren für das Leerheitsproblem für kontextfreie Sprachen: $L(G) = \varnothing$ gdw. $S \notin V^G_{prod}$.

Als nächstes beschäftigen wir uns mit ε-Produktionen und ihrer Elimination. Dazu definieren wir für eine Grammatik die Menge der Variablen, aus denen sich das leere Wort ableiten läßt. Wir nennen diese Variablen *nullierbar*.

Definition 3.10 (Nullierbare Variable)
Eine Variable A einer Grammatik G heißt nullierbar, *wenn gilt $A \Rightarrow^*_G \varepsilon$.*

Wenn eine Variable nullierbar ist, dann ist sie insbesondere produktiv (ε ist auch ein Wort über Σ).

Beobachtungen: Die Menge der nullierbaren Variablen einer Grammatik $G = (V, \Sigma, P, S)$ können wir wie folgt iterativ bestimmen:

$$V_0 = \{A \in V \mid A \to \varepsilon \in P\}$$

$$V_{i+1} = \{A \in V \mid A \to w \in P, w \in V_i^*\}$$

Wieder existiert ein $i \in \mathbb{N}$ mit $V_i = V_{i+1}$. Diese Menge nennen wir V^G_{null}. Diese Menge ist also effektiv bestimmbar und ist genau die Menge der nullierbaren Variablen von G .

Satz 3.11 (Entfernung von ε-Produktionen)

1. *Für jede kontextfreie Sprache L ist auch die Sprache $L \setminus \{\varepsilon\}$ kontextfrei.*
2. *Für jede kontextfreie Grammatik G mit $L(G) \neq \varnothing$ und $\varepsilon \notin L(G)$ gibt es eine äquivalente Grammatik G', ohne nutzlose Variablen und ohne ε-Produktionen.*

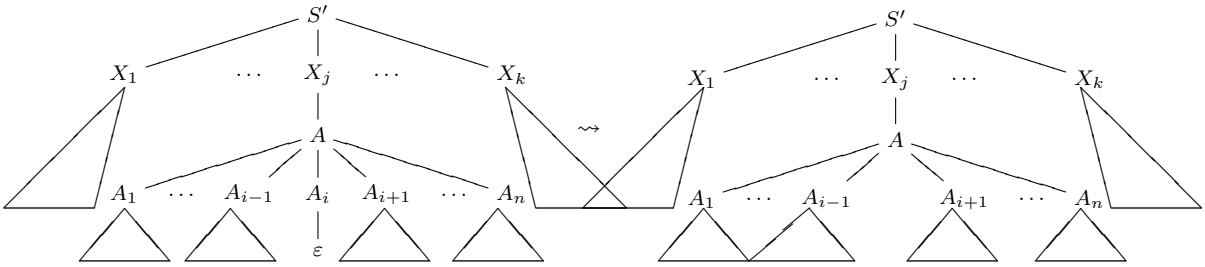
Beweis: Für eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ mit $L(G) \setminus \{\varepsilon\} \neq \varnothing$ konstruieren wir im folgenden eine Grammatik G' ohne nutzlose Variablen und ohne ε-Produktionen mit $L(G') = L(G) \setminus \{\varepsilon\}$. Daraus folgt dann Aussage 1 und 2 des Satzes.

Zum Beweis von Aussage 1 müssen wir auch den Fall $L \setminus \{\varepsilon\} = \varnothing$ (d.h. $L = \varnothing$ oder $L = \{\varepsilon\}$) untersuchen. Die Sprachen $L = \varnothing$ und $L = \{\varepsilon\}$ sind aber offensichtlich kontextfrei.

Wir fügen nun iterativ neue Produktionen zu G hinzu:

- Wenn $A \to A_1 \dots A_n$ eine Produktion ist und für ein $i \in \{1, \dots n\}$ die Variable A_i nullierbar ist, dann fügen wir die Produktion $A \to A_1 \dots A_{i-1} A_{i+1} \dots A_n$ hinzu.

Die neue Produktion simuliert eine Ableitung $A_i \Rightarrow^*_G \varepsilon$ ausgehend von der Satzform $A_1 \dots A_n$ bereits bei der Anwendung der Produktion selbst (siehe Abb. 3.3).



Redaktioneller Hinweis: Dies Graphik stimmt noch nicht ganz.

Abbildung 3.3. Simulation der Ableitung $A_i \Rightarrow_G^* \varepsilon$.

Offensichtlich verändert das Hinzufügen der Produktionen die von der Grammatik erzeugte Sprache nicht. Da die rechte Seite der hinzugefügten Produktion kürzer als die rechte Seite der Ausgangsproduktion ist, kann irgendwann keine neue Produktion mehr hinzugefügt werden. Die Grammatik, die wir dann erhalten, nennen wir G_1; sie ist äquivalent zur Grammatik G. Außerdem ist nun eine Variable A genau dann nullierbar, wenn $A \to \varepsilon$ eine Produktion von G_1 ist.
Nun löschen wir aus G_1 alle ε-Produktionen; diese Grammatik nennen wir G_2. In allen Ableitungsschritten außer dem ersten, können wir die Anwendung einer ε-Produktion von G_1 durch Anwendung einer der hinzugefügten Produktionen in einem früheren Ableitungsschritt simulieren. Nur die Anwendung einer ε-Produktion im ersten Ableitungsschritt können wir nicht simulieren. Also gilt $L(G_2) = L(G_1) \setminus \{\varepsilon\} = L(G) \setminus \{\varepsilon\}$.
Wegen $L(G_2) = L(G) \setminus \{\varepsilon\} \neq \varnothing$ können wir mit Satz 3.9 alle nutzlosen Symbole aus G_2 entfernen und erhalten eine kontextfreie Grammatik G' ohne nutzlose Variablen und ohne ε-Produktionen mit $L(G') = L(G) \setminus \{\varepsilon\}$. □

Zuletzt zeigen wir, daß man in einer Grammatik keine Kettenproduktionen benötigt:

Satz 3.12 (Entfernung von Kettenproduktionen) *Für jede nicht-leere kontextfreie Sprache, die ε nicht enthält, gibt es effektiv eine kontextfreie Grammatik ohne nutzlose Variablen, ohne ε-Produktionen und ohne Kettenproduktionen (d.h. ohne Produktionen der Form $A \to B$).*

Beweis: Gemäß Satz 3.11 existiert effektiv eine Grammatik $G = (V, \Sigma, P, S)$ ohne ε-Produktionen. Wir definieren nun:

$$P' = P \cup \{A \to \alpha \mid A \Rightarrow_G^* B \Rightarrow_G \alpha \text{ mit } |\alpha| \geq 2 \text{ oder } \alpha \in \Sigma^*\}$$

In P' fügen wir für jede Sequenz von Kettenproduktionen $A \Rightarrow_G^* B$ gefolgt von einer „Nicht-Kettenproduktion“ $B \to \alpha$ eine entsprechende abkürzende Produktion $A \to \alpha$ hinzu. Diese hinzugefügten Produktionen sind alle keine Kettenproduktionen.
Die Menge der hinzugefügten Produktionen ist endlich und effektiv bestimmbar. Außerdem enthält P' keine ε-Produktionen.

Offensichtlich gilt $L(G') = L(G)$, da die neu hinzugefügten Produktionen nur „Abkürzungen“ sind. Nun löschen wir aus G' die Kettenproduktionen, d.h. wir definieren $G'' = (V, \Sigma, P' \setminus$

$(V \times V), S)$. Die Grammatik G'' enthält weder ε-Produktionen noch Kettenproduktionen. Außerdem ist die Grammatik G'' äquivalent zu G, da in jeder Ableitung von G jede Sequenz von Kettenproduktionen durch eine der hinzugefügten Produktionen simuliert werden kann.
Wie im Beweis von Satz 3.9 können wir nun noch die nutzlosen Variablen eliminieren, ohne dabei ε-Produktionen oder Kettenproduktionen zu erzeugen. Dies ist die gesuchte Grammatik. □

1.4.2 Die Chomsky-Normalform

Bisher haben wir nur gezeigt, daß für jede kontextfreie Sprache eine äquivalente kontextfreie Grammatik ohne „Müll“ existiert. Die Form der Produktionen haben wir dabei jedoch nicht (signifikant) eingeschränkt. Nun schränken wir die Form der Produktionen noch stärker ein. In der *Chomsky-Normalform (CNF)* schränken wir die Produktionen so ein, daß die Ableitungsbäume im wesentlichen Binärbäume werden.

Satz 3.13 (Chomsky-Normalform (CNF)) *Jede ε-freie, nicht-leere kontextfreie Sprache wird von einer kontextfreien Grammatik erzeugt, deren Produktionen die Form $A \to BC$ oder $A \to a$ haben (und die keine nutzlosen Variablen enthält).*

> *Was die nutzlosen Variablen betrifft, gibt es in der Literatur verschiedene Definitionen. Manchmal wird verlangt, daß in einer Grammatik in CNF keine nutzlosen Variablen vorkommen, manchmal nicht; deshalb haben wir diese Bedingung in Klammern gesetzt.*

Beweis: Wir skizzieren hier nur die wesentliche Idee:
Gemäß Satz 3.12 gibt es für die Sprache eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ ohne nutzlose Variablen, ohne ε-Produktionen und ohne Kettenproduktionen. Aus dieser Grammatik konstruieren wir nun in zwei Schritten eine äquivalente Grammatik in Chomsky-Normalform:

1. Schritt: Für jedes Terminalzeichen $a \in \Sigma$ fügen wir eine neue Variable B_a und eine neue Produktion $B_a \to a$ hinzu.

Außerdem ersetzen wir in jeder Produktion $A \to \alpha$ mit $|\alpha| \geq 2$ jedes Zeichen $a \in \Sigma$ in α durch die neue Variable B_a.

Die so entstandene Grammatik nennen wir $G' = (V', \Sigma, P', S)$. Jede Produktion dieser Grammatik hat nun die Form $A \to a$ oder $A \to A_1 \dots A_n$ mit $n \geq 2$ und $A_i \in V'$. Offensichtlich gilt $L(G') = L(G)$.

2. Schritt: Nun ersetzen wir in G' jede Produktion der Form $A \to A_1 \dots A_n$ mit $n > 2$ durch die folgenden $n - 1$ Produktionen:

$$\begin{aligned} A &\to A_1 C_1 \\ C_1 &\to A_2 C_2 \\ C_2 &\to A_3 C_3 \\ &\ddots \\ C_{n-2} &\to A_{n-1} A_n \end{aligned}$$

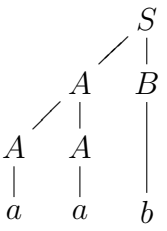
wobei $C_1, \dots, C_{n-2}$ neue Variablen sind.

□

Bemerkungen:

- Ableitungsbäume einer Grammatik in Chomsky-Normalform sind im wesentlichen vollständige Binärbäume. Bis auf die vorletzten Knoten, also die, die ein Terminalzeichen erzeugen, haben alle inneren Konten genau zwei Kinder. Dies sieht man leicht am folgenden Beispiel:

$$S \to AB \quad A \to AA \quad A \to a \quad B \to b$$



- Jede Ableitung eines Wort w der Länge n besitzt für eine kontextfreie Grammatik in CNF genau $2n-1$ Ableitungsschritte (wir sagen, daß die Ableitung die *Länge* $2n-1$ hat). Die Zahl der Ableitungsschritte setzt sich wie folgt zusammen:

  · $n-1$ Ableitungsschritte mit Regeln der Form $A \to BC$. Bei jeder Anwendung dieser Regeln wird die Satzform um ein Zeichen länger. Da wir mit dem Startsymbol S (also einer Satzform der Länge 1) beginnen, müssen wir $n-1$ Regeln der Form $A \to BC$ anwenden, um eine Satzform der Länge n zu erzeugen.

  · n Ableitungsschritte mit Regeln der Form $A \to a$, um jede Variable in ein Terminalzeichen umzuwandeln.

  Betrachten wir die Ableitung des Wortes $w = aaab$ mit der Grammatik aus dem obigen Beispiel:

$$\begin{array}{lllllllll} S & \Rightarrow & AB & \Rightarrow & AAB & \Rightarrow & AAAB & \Rightarrow & \text{3 Schritte} \\ & & aAAB & \Rightarrow & aaAB & \Rightarrow & aaaB & \Rightarrow\ aaab & \text{4 Schritte} \end{array}$$

  Diese Erkenntnis liefert uns ein Entscheidungsverfahren für die Frage $w \in L(G)$. Denn für das Wort w müssen wir „nur" die endlich vielen Ableitungen der Länge $2n-1$ durchprobieren und überprüfen, ob sie eine Ableitung für w sind. Dieses Verfahren ist effektiv; die Effizienz dieses Verfahrens ist allerdings grausam! Wir werden später ein viel effizienteres Verfahren kennenlernen.

1.4.3 Die Greibach-Normalform

An der Chomsky-Normalform gefällt uns, dass die zugehörigen Ableitungsbäume im wesentlichen Binärbäume sind. Außerdem bleibt bei der Umwandlung einer beliebigen Grammatik in eine äquivalente Grammatik in Chomsky-Normalform die Struktur des Ableitungsbaumes der ursprünglichen Grammatik erhalten.
Ein Problem der Chomsky-Normalform ist jedoch, daß in einer Ableitung zunächst keine Terminalzeichen erzeugt werden. Die Konstruktion einer Ableitung für ein Wort wird deshalb nicht vom abzuleitenden Wort getrieben. Zu diesem Zweck sind Regeln der Form $A \to a\alpha$ mit $a \in \Sigma$

und$\alpha \in V^*$ besser geeignet, da man sich bei der Wahl der nächste Regel am nächsten Zeichen a des Wortes orientieren kann. Dabei wird in jedem Ableitungsschritt genau ein Terminalzeichen erzeugt. Für eine Grammatik mit Produktionen der obigen Form sagen wir, die Grammatik sei in *Greibach-Normalform*.

Trotz dieser praktisch klingenden Motivation ist die Greibach-Normalform nicht von großem praktischen Nutzen. Sie erleichtert uns jedoch die Herstellung des Zusammenhangs zwischen kontextfreien Sprachen und Kellerautomaten.

Vorbereitende Überlegungen Für den Beweis der Existenz der Greibach-Normalform benötigen wir einige weitere äquivalente Umformungen von Grammatiken.

Beispiel 3.5
Die erste Produktion der Grammatik auf der linken Seite ist nicht in Greibach-Normalform. Offensichtlich wird aber das B der rechten Seite dieser Produktion entweder durch b oder bA ersetzt; wir können also die Produktion $A \to BC$ durch zwei Produktionen $A \to bC$ und $A \to bAC$ ersetzen, wie dies in der rechten Grammatik dargestellt ist.

$$\begin{array}{lcl} A \to BC & & A \to bC \mid aAC \\ B \to b \mid bA & \rightsquigarrow & B \to b \mid aA \\ C \to c \mid cB & & C \to c \mid cB \end{array}$$

Durch diese Transformation werden zwei Ableitungsschritte der linken Grammatik $A \Rightarrow BC \Rightarrow bC$ bzw. $A \Rightarrow BC \Rightarrow bAC$ zu einem Ableitungsschritt $A \Rightarrow bC$ bzw. $A \Rightarrow bAC$ zusammengefaßt.

Diese Zusammenfassung von Ableitungsschritten durch Substitution einer Variablen durch alle ihre rechten Seiten können wir noch etwas allgemeiner wie folgt formulieren:

Lemma 3.14 (Substitution auf der rechten Seite einer Produktion) *Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik, $A \to \alpha B \beta$ eine Produktion von G mit $\alpha, \beta \in (V \cup \Sigma)^*$ und $B \in V$. Sei $R_B = \{\gamma \in (V \cup \Sigma)^* \mid B \to \gamma \in P\}$ die Menge aller rechten Seiten von B und $P' = (P \setminus \{A \to \alpha B \beta\}) \cup \{A \to \alpha \gamma \beta \mid \gamma \in R_B\}$ und $G' = (V, \Sigma, P', S)$. Dann gilt $L(G') = L(G)$.*

Beweis: Wir beweisen die Gleichheit, indem wir die Inklusion in beide Richtungen zeigen:

„$\subseteq$“: Wir zeigen, daß mit $w \in L(G')$ auch $w \in L(G)$ gilt. Da G alle Produktionen aus G' bis auf die Produktionen $A \to \alpha\gamma\beta$ mit $\gamma \in R_B$ enthält, genügt es zu zeigen, daß wir diese Produktion in G simulieren können: $A \Rightarrow_G \alpha B \beta \Rightarrow_G \alpha\gamma\beta$.

„$\supseteq$“: Wir zeigen, daß mit $w \in L(G)$ auch $w \in L(G')$ gilt. Dazu betrachten wir einen Ableitungsbaum für w in G. Wenn in dem Baum die Produktion $A \to \alpha B \beta$ nicht (als Verzweigung) vorkommt, ist der Baum auch ein Ableitungsbaum in G' (weil G' alle Produktionen von G außer $A \to \alpha B \beta$ enthält), und damit $w \in L(G')$.

Wir betrachten nun einen Ausschnitt eines Baumes, in dem die Produktion $A \to \alpha B \beta$ als Verzweigung vorkommt. Ein solcher Teilausschnitt ist auf der linken Seite von Abb. 3.4 dargestellt. Dabei muß für γ eine Produktion $B \to \gamma$ existieren. Also ist $A \to \alpha\gamma\beta$

Abbildung 3.4. Transformation des Ableitungsbaumes

eine Produktion von G'; wir können also den Ausschnitt auf der linken Seite durch den Ausschnitt auf der rechten Seite von Abb. 3.4 ersetzen. Wenn wir alle Vorkommen von $A \to \alpha B \beta$ in dem Ableitungsbaum in G auf diese Weise ersetzen, erhalten wir einen Ableitungsbaum für w in G'. Also gilt $w \in L(G')$.

□

In Beispiel 3.5 hat die Substitution auf der rechten Seite einer Produktion sofort zu einer Grammatik in Greibach-Normalform geführt. Dies ist leider nicht immer der Fall. Insbesondere bei Grammatiken mit *linksrekursiven* Produktionen, d.h. mit Produktionen der Form $A \to A\alpha$, führt dies nicht zum Ziel. Wir müssen deshalb linksrekursive Produktionen eliminieren.
Auf der linken Seite von Abb. 3.5 ist eine Grammatik mit linksrekursiven Produktionen für A dargestellt. Darunter ist ein Ableitungsbaum in G schematisch dargestellt. Auf der rechten Seite ist eine äquivalente Grammatik angegeben, in der die linksrekursiven Produktionen für A durch *rechtsrekursive* Produktionen für eine neue Variable B simuliert werden. Offensichtlich erzeugen die Grammatiken dieselben Sprachen. Der Ableitungsbaum wird durch diese Transformation „rotiert“.
Diesen Sachverhalt formalisieren wir nun (dabei entspricht A_α genau der Menge der α_i in Abb. 3.5 und A_β der Menge der β_i):

Lemma 3.15 (Elimination linksrekursiver Produktionen) *Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik, sei A eine Variable der Grammatik und seien die Mengen A_α und A_β wie folgt definiert:*

$$\begin{aligned} A_\alpha &= \{\alpha \in (V \cup \Sigma)^* \mid A \to A\alpha \in P\} \\ A_\beta &= \{\beta \in (V \cup \Sigma)^* \mid A \to \beta \in P, \nexists \beta' : \beta = A\beta'\} \end{aligned}$$

Wir definieren nun eine Menge von Produktionen

$$\begin{aligned} P' = \quad & P \setminus (\{A\} \times (V \cup \Sigma)^*) && \left.\vphantom{x}\right\} \text{ löscht alle } A\text{-Produktionen} \\ & \cup \{A \to \beta \mid \beta \in A_\beta\} && \left.\vphantom{x}\right\} \\ & \cup \{A \to \beta B \mid \beta \in A_\beta\} && \left.\vphantom{x}\right\} \text{ rotiert } A\text{-Produktionen} \\ & \cup \{B \to \alpha \mid \alpha \in A_\alpha\} && \left.\vphantom{x}\right\} \\ & \cup \{B \to \alpha B \mid \alpha \in A_\alpha\} && \left.\vphantom{x}\right\} \text{ zusätzliche } B\text{-Produktionen} \end{aligned}$$

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n$$
$$A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

$$A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$
$$A \to \beta_1 B \mid \beta_2 B \mid \ldots \mid \beta_m B$$
$$B \to \alpha_1 B \mid \alpha_2 B \mid \ldots \mid \alpha_n B$$
$$B \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

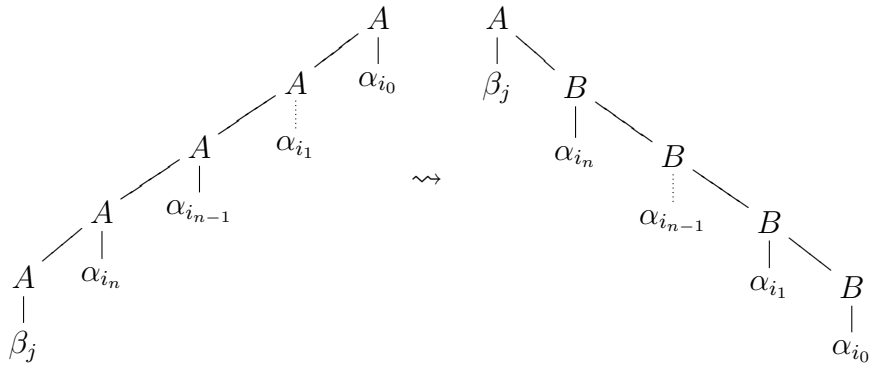


Abbildung 3.5. „Rotation"des Ableitungsbaumes

wobei $B \notin (V \cup \Sigma)^*$ eine neue Variable ist. Für die Grammatik $G' = (V \cup \{B\}, \Sigma, P', S)$ gilt dann $L(G) = L(G')$.

Beweis: Formalisierung der Vorüberlegung (vgl. Abb. 3.5). □

Bemerkung: Die Struktur der Ableitungsbäume für dasselbe Wort unterscheiden sich in G und in G' deutlich!

Überführung in GNF: Die Überführung einer Grammatik in Greibach-Normalform ist eine geschickte Kombination der Transformationen aus Lemma 3.14 und Lemma 3.15. Insgesamt ähnelt dieses Verfahren, dem Verfahren zum Lösen eines linearen Gleichungssystems (Gaußelimination und Rückwärtssubstitution). Wir erklären dieses Verfahren zunächst anhand eines Beispiels; das Verfahren selbst wird dann im Beweis von Satz 3.16 angegeben.

$$\begin{array}{lcl} S & \to & AB \\ A & \to & AA \\ A & \to & a \\ B & \to & b \\ B & \to & S \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} S & \to & AB \\ A & \to & a \\ A & \to & aC \\ B & \to & b \\ B & \to & SA \end{array} \qquad \begin{array}{lcl} C & \to & A \\ C & \to & AC \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} S & \to & AB \\ A & \to & a \\ A & \to & aC \\ B & \to & b \\ B & \to & aBA \\ B & \to & aCBA \end{array} \qquad \begin{array}{lcl} C & \to & A \\ C & \to & AC \end{array}$$

(a) (b) (c)

Abbildung 3.6. Überführung einer Grammatik in Greibach-Normalform: Schritt 1

Beispiel 3.6
Wir werden im folgenden zeigen, wie man die Grammatik aus Abb. 3.6 (a) in Greibach-Normalform überführt:

1. Schritt: Zunächst legen wir eine Reihenfolge auf den Variablen der Grammatik fest: $S < A < B$ (wir haben die Regeln der Grammatik bereits gemäß dieser Reihenfolge sortiert). Im ersten Schritt sorgen wir nun dafür, daß auf der rechten Seite aller Regeln einer Variablen X als erstes Symbol entweder ein Terminalzeichen oder eine Variable Y mit $Y > X$ vorkommt.

Dazu gehen wir der Reihe nach die Regeln der Variablen S, A und B durch:

S: Die (einzigen) Regel für S ist bereits in der gewünschten Form (das erste Symbol der rechten Seite ist A und es gilt $A > S$).

A: Die Regel $A \to AA$ hat noch nicht die gewünschte Form, da $A \not> A$ (die Regel ist linksrekursiv). Deshalb führen wir eine Rotation der Regeln für A (Lemma 3.15) durch mit $A_\alpha = \{A\}, A_\beta = \{a\}$ und der neuen Variablen C. Dann erhalten wir die Grammatik aus Abb. 3.6 (b).

In dieser Grammatik haben nun auch die Regeln für A die gewünschte Form.

B: Die Regel $B \to SA$ hat noch nicht die gewünschte Form, da $S < B$. Um diese Regel in die gewünschte Form zu bringen, substituieren wir S in dieser Regel durch seine rechten Seiten (Lemma 3.14). Wir erhalten also die Regel $B \to ABA$. Diese Regel ist leider immer noch nicht in der gewünschten Form[2]. Wir substituieren also nochmals (diesmal die rechten Seiten von A) für das erste A und erhalten die zwei Regeln $B \to aBA$ und $B \to aCBA$. Nach diesen beiden Schritten erhalten wir die Grammatik aus Abb. 3.6 (c).

Diese Grammatik ist bereits in der Form, die wir im 1. Schritt erreichen wollten. Insbesondere sind die ersten Zeichen der rechten Seiten von B bereits Terminalzeichen.

Es hätte auch passieren können, daß nach der Substitution von A auf der rechten Seite von B eine Regel der Form $B \to B\alpha$ entsteht. Diese Linksrekursion hätten wir wieder durch Rotation beseitigen müssen.

2. Schritt: Im zweiten Schritt sorgen wir dafür, daß in allen Regeln für S, A und B das erste Zeichen auf der rechten Seite ein Terminalzeichen ist. Für die letzte Variable muß dies nach dem ersten Schritt bereits der Fall sein. Für die anderen Variablen erreichen wir dies, indem wir von unten nach oben (also in umgekehrter Reihenfolge der Variablen) substituieren.

Im Beispiel müssen wir nur noch das erste A in der rechten Seite von S durch Substitution beseitigen. Wir erhalten die Grammatik aus Abb. 3.7 (d).

3. Schritt Als letztes müssen wir noch die Regeln für die neu hinzugefügten Variablen betrachten. Im Beispiel müssen wir also die Variable A in den rechten Seiten von C substituieren und erhalten die Grammatik aus Abb. 3.7 (e).

[2] Die Regel $B \to ABA$ ist aber schon etwas „besser“, als $B \to SA$, da $A > S$.

$$
\begin{array}{lllll}
 & S \to aB & & S \to aB & \\
 & S \to aCB & & S \to aCB & \\
 & A \to a \quad C \to A & & A \to a & C \to a \\
\rightsquigarrow & A \to aC \quad C \to AC & \rightsquigarrow & A \to aC & C \to aC \\
 & B \to b & & B \to b & C \to aC \\
 & B \to aBA & & B \to aBA & C \to aCC \\
 & B \to aCBA & & B \to aCBA &
\end{array}
$$

(d) (e)

Abbildung 3.7. Überführung einer Grammatik in Greibach-Normalform: Schritte 2 und 3

Bei schematischer Anwendung der Substitution entsteht die Regel $C \to aC$ zweimal. Eine davon können wir natürlich weglassen.

Insgesamt haben wir also in drei Schritten eine Grammatik in Greibach-Normalform erzeugt. Der erste Schritt ähnelt der Gaußelimination und der zweite Schritt der Rückwärtssubstitution.

Ganz allgemein gilt:

Satz 3.16 (Greibach-Normalform (GNF)) *Jede ε-freie kontextfreie Sprache kann durch eine kontextfreie Grammatik in* Greibach-Normalform *erzeugt werden, d.h. durch eine Grammatik, deren Produktionen die Form $A \to a\alpha$ mit $a \in \Sigma$ und $\alpha \in V^*$ haben.*

Beweis: Sei L eine kontextfreie Sprache. Wir können im folgenden annehmen, dass L nicht leer ist, denn für $L = \varnothing$ erfüllt die Grammatik $S \to aS$ die Anforderungen des Satzes. Nach Satz 3.13 gibt es also eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ in Chomsky-Normalform mit $L = L(G)$. Aus dieser Grammatik konstruieren wir in drei Schritten eine äquivalente Grammatik in Greibach-Normalform. Da wir bei der Konstruktion nur Transformationen einsetzen, die die erzeugte Sprache nicht verändern, ist die konstruierte Grammatik äquivalent zu G. Um zu sehen, daß die erzeugte Grammatik in Greibach-Normalform ist, geben wir für die folgenden Konstruktionsvorschriften jeweils die Schleifeninvarianten an.

Es ist sicher hilfreich, sich die folgenden Schritte anhand des Beispiels 3.6 zu veranschaulichen.

1. Schritt Sei $V = \{A_1, \ldots, A_n\}$.
Ziel ist es, die Produktionen der Grammatik so umzuformen, daß am Ende gilt: Wenn $A_i \to A_j\alpha$ eine Produktion ist, dann gilt $j > i$.

for $i = 1, \ldots, n$
Schleifeninvariante
- jede rechte Seite einer A_k Produktion hat die Form $a\alpha$ oder $VV\alpha$
- für jede Produktionen $A_k \to A_j\alpha$ mit $k < i$ gilt bereits $j > k$

while (es existiert eine Produktion $A_i \to A_j\alpha$ mit $j < i$) **do**
lösche $A_i \to A_j\alpha$ /* Subst. mit Lemma 3.14 */
füge $A_i \to \gamma\alpha$ für alle $\gamma \in R_{A_j}$ hinzu
od
/* Für jede Produktion der Form $A_i \to A_j\alpha$ gilt jetzt $j \geq i$ */

/* Produktionen der Form $A_i \to A_i\alpha_j$ eliminieren wir mit Lemma 3.15:
α_i und β_j wie in Lemma 3.15 und B_i neue Variable */
Streiche $A_i \to A_i\alpha_j$ und
füge $A_i \to \beta_j$, $A_i \to \beta_j B_i$, $B_i \to \alpha_j$ und $B_i \to \alpha_j B_i$ hinzu.
end

Die innere while-Schleife terminiert, da aufgrund der Schleifeninvariante der for-Schleife in allen $\gamma = A_k\gamma'$ gilt $k > j$; d.h. die führende Variable auf der rechten Seite der neuen Produktion ist größer als die der ersetzten Produktion.

Nach Schritt 1 gilt nun:

- Jede Produktion für A_i hat jetzt die Form $A_i \to A_j\alpha$ mit $j > i$ oder $A_i \to a\alpha$. Für die letzte Variable A_n können die Produktionen nur die Form $A_n \to a\alpha$ haben, da es keine Variable A_j mit $j > n$ gibt.
- Jede Produktion für B_i hat jetzt die Form $B_i \to A_j\alpha$ oder $B_i \to a\alpha$.

2. Schritt

Ziel ist es, alle A_i-Produktionen in die Form $A_i \to a\alpha$ zu bringen.

for $i = n-1, n-2, \ldots, 1$
Schleifeninvariante
jede A_k Produktion mit $k > i$ hat die Form $A_k \to a\alpha$
while (es existiert eine Produktion $A_i \to A_j\alpha$) **do**
lösche $A_i \to A_j\alpha$ /* Subst. mit Lemma 3.14 */
füge $A_i \to \gamma\alpha$ für alle $\gamma \in R_{A_j}$ hinzu
od
end

Nach Schritt 2 gilt nun:

- Jede Produktion für A_i hat jetzt die Form $A_i \to a\alpha$.
- Jede Produktion für B_i hat (bereits nach dem ersten Schritt) die Form $B_i \to A_j\alpha$ oder $B_i \to a\alpha$.

3. Schritt

Ziel ist es, jede B_i-Produktion in die Form $B_i \to a\alpha$ zu bringen.
while (es existiert eine Produktion $B_i \to A_j\alpha$) **do**
lösche $B_i \to A_j\alpha$ /* Subst. mit Lemma 3.14 */
füge $B_i \to \gamma\alpha$ für alle $\gamma \in R_{A_j}$ hinzu
od

Nach Schritt 3 sind nun sowohl die A_i-Produktionen als auch die B_i-Produktionen in Greibach-Normalform. □

Bemerkungen:

1. Bei der Konstruktion können sehr viele Produktionen entstehen ($\sim n^n$). Dies ist nicht praktikabel. Es existieren aber bessere Verfahren, bei denen nur kubisch viele Produktionen ($\sim n^3$) entstehen können.

2. Die Struktur der Ableitungsbäume wird durch die Konstruktion stark verändert (wg. der Rotation aus Lemma 3.15). Damit ist diese Überführung einer Grammatik in Greibach-Normalform ungeeignet für den Compilerbau.

3. Das Konstruktionsverfahren ähnelt dem Verfahren zur Lösung linearer Gleichungssysteme (siehe lineare Algebra und Gleichungssysteme für reguläre Sprachen).

4. Die Ableitung eines Satzes einer Grammatik in Greibach-Normalform hat genau die Länge des abgeleiteten Satzes, da in jedem Ableitungsschritt genau ein Terminalzeichen erzeugt wird.

2 Kellerautomaten

Wir werden nun ein Automatenmodell für die kontextfreien Sprachen angeben und näher untersuchen: die *Kellerautomaten*. Ähnlich wie bei endliche Automaten für reguläre Sprachen, gibt es verschiedene Varianten von Kellerautomaten; im Gegensatz zu den endlichen Automaten sind diese Automaten nicht alle äquivalent zueinander. Insbesondere sind die deterministischen Kellerautomaten nicht so ausdrucksmächtig wir die nicht-deterministischen Kellerautomaten. Da die deterministischen Kellerautomaten besonders praxisrelevant sind, werden wir die von ihnen erkannten Sprachen später in Abschnitt 5 noch genauer untersuchen.

2.1 Motivation und Definition

Zunächst überlegen wir uns, wie ein Automat aussehen könnte, der die Sprache $a^n b^n$ akzeptiert. Offensichtlich muß ein Automat für diese Sprache „zählen“ können, da er überprüfen muß, ob die Zahl der a's und b's übereinstimmt. Der Automat muß also einen beliebig großen Speicher besitzen. Der Automat könnte das Wort einlesen, und dabei jedes gelesene Zeichen a auf einen *Keller* legen. Wenn der Automat auf das erste Zeichen b stößt, entfernt er für jedes gelesene b ein a vom Keller. Wenn am Ende alle b's der Eingabe gelesen wurden, und kein a mehr im Keller steht, dann akzeptiert der Automat das Wort. Mit Hilfe des Kellers können wir also zählen; tatsächlich können wir einen Keller als Verallgemeinerung eines Zählers auffassen.
Natürlich muß der Automat auch sicherstellen, daß kein a mehr im Eingabewort vorkommt, wenn bereits ein b gelesen wurde. Diese können wir wie bei endlichen Automaten sicherstellen. Ein Kellerautomat ist also ein um einen Keller erweiterter endlicher Automat:

Definition 3.17 (Kellerautomat) *Ein Kellerautomat A über einen Alphabet Σ besteht aus*

- *einer endlichen Menge Q von Zuständen mit $Q \cap \Sigma = \varnothing$,*

- *einem* Startzustand $q_0 \in Q$,

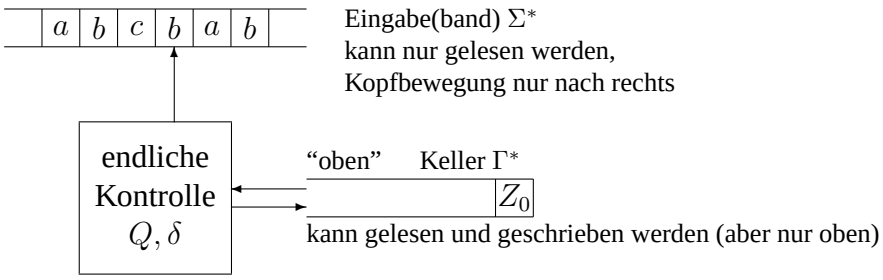


Abbildung 3.8. Schematische Darstellung eines Kellerautomaten.

- *einem* Kelleralphabet Γ *mit* $Q \cap \Gamma = \varnothing$ *(*$\Gamma = \Sigma$ *ist möglich aber nicht nötig)*
- *einem* Anfangssymbol $Z_0 \in \Gamma$
- *einer Menge* F *von* Endzuständen
- *eine* Übergangsrelation $\delta \subseteq (Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$

Wir schreiben $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$.

> *Die Einschränkungen* $Q \cap \Sigma = \varnothing$ *und* $Q \cap \Gamma = \varnothing$ *wäre nicht nötig; sie vereinfachen aber später die Definition der Konfigurationen eines Kellerautomaten und schränken die akzeptierbaren Spracjen nicht ein.*

In Abb. 3.8 ist ein Kellerautomat schematisch dargestellt.

Beispiel 3.7
Wir geben einen Kellerautomaten an, der die Palindrome über $\Sigma = \{a, b\}$ erkennt. Der Kellerautomat arbeitet wie folgt: Zunächst werden Eingabezeichen gelesen und auf den Keller geschrieben („gepusht“). Irgendwann wird „geraten“, dass die Mitte des Wortes erreicht ist (nicht-deterministisch). Ab diesem Zeitpunkt wird überprüft, ob das oberste Symbol des Kellers mit dem aktuellen Eingabezeichen übereinstimmt; wenn ja, wird das oberste Symbol vom Keller entfernt („gepoppt“); wenn nein, bleibt der Automat stehen (und akzeptiert das Eingabe Wort nicht).
Wir formalisieren nun diese Idee:

- $Q = \{q_0, q_1\}$
- $\Gamma = \Sigma \cup \{c\}$
- $Z_0 = c$
- $(F = \varnothing)$
- Die Produktionen:

$$
\begin{aligned}
\delta \;=\; & \{((q_0, x, y), (q_0, xy)) \mid x \in \Sigma, y \in \Gamma\} \cup && \text{(1) Nächstes Eingabezeichen } x \text{ pushen}\\
& \{((q_0, \varepsilon, y), (q_1, y)) \mid y \in \Gamma\} \cup && \text{(2) Entscheidung Mitte (gerade Länge)}\\
& \{((q_0, x, y), (q_1, y)) \mid x \in \Sigma, y \in \Gamma\} \cup && \text{(3) Entscheidung Mitte (ungerade Länge)}\\
& \{((q_1, x, x), (q_1, \varepsilon)) \mid x \in \Sigma\} \cup && \text{(4) Keller mit der Eingabe vergleichen}\\
& \{((q_1, \varepsilon, c), (q_1, \varepsilon))\} && \text{(5) Anfangssymbol vom Keller löschen}
\end{aligned}
$$

Nun betrachten wir einige *Berechnungen* dieses Kellerautomaten, wobei die Nummern in Klammern auf die Zeile in der Definition der Übergangsrelation verweist, die den entsprechenden Übergang ermöglicht.

Eingabe	abba		Eingabe	aba	
	q_0c			q_0c	
	aq_0ac	(1)		aq_0ac	(1)
	abq_0bac	(1)		abq_0ac	(3)
	abq_1bac	(2)		$abaq_1c$	(4)
	$abbq_1ac$	(4)		$abaq_1$	(5)
	$abbaq_1c$	(4)			
	$abbaq_1$	(5)			

Die *Berechnungen* eines Kellerautomaten ist eine Folge von Konfigurationen des Kellerautomaten. Dabei besteht eine Konfiguration aus dem bisher gelesenen Eingabewort (links), dem aktuellen Zustand (mitte) und dem aktuellen Kellerinhalt (rechts).

Definition 3.18 (Konfiguration, Nachfolgekonfiguration) *Sei $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ ein Kellerautomat. Ein Wort $\gamma \in \Sigma^* Q \Gamma^*$ heißt* Konfiguration *von A. Die Konfiguration $\gamma = q_0 Z_0$ heißt* Startkonfiguration *von A.*
Für eine Konfiguration $\gamma = wqy\alpha$ mit $w \in \Sigma^*$, $q \in Q$, $y \in \Gamma$, $\alpha \in \Gamma^*$ und $x \in \Sigma \cup \{\varepsilon\}$, $\beta \in \Gamma^*$ mit $((q, x, y), (p, \beta)) \in \delta$ heißt $\gamma' = wxp\beta\alpha$ eine Nachfolgekonfiguration *von γ in A. Wir schreiben dann $\gamma \vdash_A \gamma'$.*

> *Die Startkonfiguration $q_0 Z_0$ können wir auch durch $\varepsilon q_0 Z_0$ notieren. Dann kommt die Aufteilung der Konfiguration in die bisher gelesene Eingabe, den Zustand und den Keller auch in der Startkonfiguration besser zum Ausdruck.*

Bemerkung: In der Literatur wird die Konfiguration (fast) immer anders definiert: Es wird nicht die bisher gelesene Eingabe aufgenommen, sondern die noch zu lesende Eingabe. In unserer Definition weichen wir davon ab, damit wir später den Zusammenhang zu den kontextfreien Sprachen einfacher herstellen können. Tatsächlich müßte man weder die bisher gelesene noch die noch zu lesende Eingabe in die Definition der Konfiguration aufnehmen. Das ist aber für die Definition der akzeptierten Sprache weniger zweckmäßig.

Die akzeptierte Sprache eines Kellerautomaten können wir nun mit Hilfe der Berechnungen definieren. Allerdings gibt es zwei verschiedene Varianten: Akzeptierung über Endzustände und Akzeptierung mit leerem Keller.

Definition 3.19 (Akzeptierte Sprache) *Sei $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ ein Kellerautomat.*

1. *Wir nennen $L(A) = \{w \in \Sigma^* \mid q_0 Z_0 \vdash_A^* wq\alpha$ mit $q \in F\}$ die (über die Endzustände)* akzeptierte Sprache *von A.*

2. *Wir nennen $N(A) = \{w \in \Sigma^* \mid q_0 z_0 \vdash_A^* wq$ mit $q \in Q\}$ die* mit leerem Keller akzeptierte Sprache *von A.*

Beispiel 3.8
In Beispiel 3.7 haben wir die Akzeptierung mit leeren Keller benutzt; deshalb war die Definition der Endzustandsmenge unerheblich. Wenn wir den Automaten nun zur Akzeptierung über Endzuständen benutzen wollen, müßten wir den Automaten leicht modifizieren:
Beim Löschen des letzten Kellersymbols c geht der Automat in einen neuen Endzustand q_2 über. Die fünfte Regel unseres Palindrom-Beispiels lautet dann: $\{((q_1, \varepsilon, c), (q_2, \varepsilon))\}$. Die Menge der Endzustände ist dann $F = \{q_2\}$.

Tatsächlich können wir jeden Automaten, der eine Sprache mit leerem Keller akzeptiert, in einen Automaten umbauen, der die Sprache über Endzustände akzeptiert, und umgekehrt.

Satz 3.20 (Äquivalenz der Akzeptierung über Endzustände und mit leeren Keller)

1. *Für jeden Kellerautomaten A gibt es (effektiv) einen Kellerautomaten A' mit $L(A') = N(A)$.*

2. *Für jeden Kellerautomaten A' gibt es (effektiv) einen Kellerautomaten A mit $N(A) = L(A')$.*

Die Klasse der Sprachen, die von einen Kellerautomaten mit leerem Keller akzeptiert akzeptiert werden, ist gleich der Klasse der Sprachen, die von einem Kellerautomaten über Endzustände akzeptiert werden.

Beweis: Wir skizzieren hier nur die Ideen für den Beweis der beiden Aussagen:

1. Wir konstruieren aus dem Kellerautomaten $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ einen Automaten A' mit $L(A') = N(A)$:

   - A' ersetzt im ersten Berechnungsschritt das Kellersymbol Z_0 durch Z_0X_0, wobei $X_0 \notin \Gamma$ ein neues Kellersymbol ist.
   - Dann simuliert A' den Automaten A.
   - Wenn in A' das Symbol X_0 das oberstes Kellersymbol ist (d.h. wenn die Berechnung von A mit leerem Keller beendet ist), dann löscht A' das Symbols X_0 aus dem Keller und geht in einen neuen Zustand $q_f \notin Q$ über, den wir als Endzustand auszeichnen $F' = \{q_f\}$.

   Durch diese Simulation von A geht A' genau dann in den Endzustand q_f über, wenn A das Wort mit leerem Keller akzeptiert. Deshalb gilt $L(A') = N(A)$.

2. Wir konstruieren nun aus dem Kellerautomaten A' einen Automaten A mit $N(A) = L(A')$. Zunächst die grobe Idee:

   - Der Automat A simuliert den Automaten A'.
   - Wenn A in der Simulation einen Endzustand von A' erreicht, dann kann sich A' nicht-deterministisch dafür entscheiden, den Keller vollständig zu löschen (und damit das bisher gelesene Wort zu akzeptieren).

Also wird jedes Wort, das A' über Endzustände akzeptiert, von A mit leerem Keller akzeptiert.

Es gibt allerdings noch ein Problem: Wenn A bei der Simulation von A' den leeren Keller erreicht, sich aber A' nicht in einem Endzustand befindet, wird A das entsprechende Wort akzeptieren, obwohl es von A' nicht akzeptiert wird.

Dieses Problem läßt sich wie folgt lösen: Im ersten Schritt erzeugt A aus dem Startsymbol Z_0 den Kellerinhalt Z_0X_0 mit einem neuen Kellersymbol X_0. Da X_0 ein neues Kellersymbol ist, wird bei der Simulation von A' durch A der Keller nie leer werden. Der Keller von A wird also nur dann leer, wenn sich A in einem Endzustand von A' nicht-deterministisch für das Löschen des Kellers entscheidet. Also gilt dann $N(A) = L(A)$.

□

Bemerkung: Satz 3.20 gilt nicht für deterministisch Kellerautomaten. Überlegen Sie sich dazu, ob es einen deterministischen Kellerautomaten geben kann, der sowohl das Wort w als auch das Wort wv (mit $v \neq \varepsilon$) mit leerem Keller akzeptiert?

Die Antwort auf diese Frage gibt es in der Übung (Blatt 8, Aufgabe 4).

Dazu müssen wir natürlich erst einmal definieren, wann ein Kellerautomat deterministisch ist:

Definition 3.21 (Deterministischer Kellerautomat) *Ein Kellerautomat* $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ *heißt* deterministisch, *wenn für alle* $q, q_1, q_2 \in Q$, $x \in \Gamma$, $a \in \Sigma$ *und* $a' = a$ *oder* $a' = \varepsilon$ *und* $\alpha_1, \alpha_2 \in \Gamma^*$ *mit* $((q, a, x), (q_1, \alpha_1)) \in \delta$ *und* $((q, a', x), (q_2, \alpha_2)) \in \delta$ *gilt:* $a = a'$, $q_1 = q_2$ *und* $\alpha_1 = \alpha_2$.

Man sieht leicht, dass für einen deterministischen Kellerautomaten für jede Konfiguration γ höchstens eine Nachfolgekonfiguration existiert.

2.2 Kellerautomaten und kontextfreie Sprachen

Der zentrale Satz über Kellerautomaten besagt, daß die von (*nicht-deterministischen*) Kellerautomaten akzeptierten Sprachen genau die kontextfreien Sprachen sind. Wegen Satz 3.20 reicht es, dies für die Kellerautomaten zu zeigen, die eine Sprache mit leerem Keller akzeptieren.
Diese Aussage formulieren wir in zwei separaten Sätzen (entsprechend der beiden Richtungen der Äquivalenz).

Satz 3.22 *Jede kontextfreie Sprache wird von einem Kellerautomaten (mit leerem Keller) akzeptiert.*

Beweis: Wir führen den Beweis nur für kontextfreie Sprachen L, die ε nicht enthalten (eine Erweiterung auf Sprachen mit $\varepsilon \in L$ ist aber nicht schwer). Wir können also nach Satz 3.16 davon ausgehen, daß es eine Grammatik $G = (V, \Sigma, P, S)$ mit $L(G) = L$ in Greibach-Normalform gibt.

Aus dieser Grammatik G konstruieren wir nun einen Kellerautomaten A, der die Sprache mit leerem Keller akzeptiert:
$A = (\{q\}, \Sigma, V, \delta, q, S, \varnothing)$ mit $\delta = \{((q, a, B), (q, \alpha)) \mid B \to a\alpha \in P\}$, d.h. jede Regel der Grammatik wird in einen entsprechenden Übergang des Automaten übersetzt. Dann gilt offensichtlich:

$$\underbrace{w}_{\text{gelesene Eingabe}}\ q\ \underbrace{B\gamma}_{\text{Keller}} \quad \vdash_A \quad waq\alpha\gamma$$

$$\Updownarrow$$

$$\underbrace{w}_{\substack{\text{bereits}\\ \text{erzeugtes Wort}\\ \text{(ohne Variablen)}}} B\gamma \quad \overset{L}{\Rightarrow}_G \quad wa\alpha\gamma$$

Per Induktion über die Ableitung kann man damit zeigen:

$$\begin{array}{ccc} wq\gamma & \vdash_A^* & vq\gamma' \\ & \Updownarrow & \\ w\gamma & \overset{L}{\Rightarrow}_G^* & v\gamma' \end{array}$$

Die Berechnung des Automaten auf einem Wort entspricht genau der Linksableitung des Wortes in der Grammatik. Damit die Korrespondenz zwischen einer Berechnung des Automaten und der Linksableitung der Grammatik so offensichtlich, ist haben wir die Konfigurationen etwas anders als üblich definiert.

Insbesondere gilt $w \in N(A)$ gwd. $qS \vdash_A^* wq$ gdw. $S \Rightarrow_G^* w$ gdw. $w \in L(G)$. Also gilt $N(A) = L(G) = L$. □

Bemerkung: Die Berechnung des Automaten A liefert uns eine Linksableitung für das gelesene Wort. Da der Automat im allgemeinen nicht deterministisch ist, hilft uns das in der Praxis aber relativ wenig (später mehr dazu).

Nun zur umgekehrten Richtung: Ist jede von einem Kellerautomaten akzeptierte Sprache kontextfrei?

Satz 3.23 *Jede von einem Kellerautomaten akzeptierte Sprache ist kontextfrei.*

Beweis: Nach Satz 3.20 reicht es, einen Kellerautomaten zu betrachten, der die Sprache mit leerem Keller akzeptiert: also $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ mit $N(A) = L$. Für diesen Automaten müssen wir jetzt eine kontextfreie Grammatik G mit $L(G) = N(A)$ konstruieren.

Problem: Da A im allgemeinen mehr als einen Zustand besitzt, können wir den Beweis von Satz 3.22 nicht einfach „umdrehen“. Wir müssen sowohl den aktuellen Zustand als auch den Kellerinhalt in den Variablen der Grammatik codieren. Deshalb sind die Variablen der Grammatik Tripel: $[q, B, q']$ mit $q, q' \in Q$ und $B \in \Gamma$.

Wir definieren $G = (V, \Sigma, P, S)$ nun wie folgt, wobei S ein neues Symbol ist, das als Startsymbol dient:

$$\begin{aligned} V &= Q \times \Gamma \times Q \cup \{S\} \\ P &= \{S \to [q_0, Z_0, q] \mid q \in Q\} \cup \\ &\quad \{[q, B, q_{n+1}] \to a[q_1, B_1, q_2][q_2, B_2, q_3] \dots [q_n, B_n, q_{n+1}] \mid \\ &\qquad q, q_1, q_2, \dots\ q_{n+1} \in Q, a \in \Sigma \cup \{\varepsilon\}, B_1, \dots, B_n \in \Gamma, \\ &\qquad ((q, a, B)(q_1, B_1 \dots B_n)) \in \delta\} \end{aligned}$$

Für $n = 0$ sieht die entsprechende Produktion wie folgt aus: $[q, B, q_1] \to a$.

Offensichtlich gilt:

$$\begin{array}{ccc} [q, Z, q_{n+1}] & \Rightarrow_G & a[q_1, Z_1, q_2][q_2, Z_2, q_3] \dots [q_n, Z_n, q_{n+1}] \\ & \Updownarrow & \\ qZ & \vdash_A & aq_1 Z_1 \dots Z_n \end{array}$$

Per Induktion kann man zeigen: Wenn $[q, Z, q_{n+1}] \overset{L}{\Rightarrow}{}^*_G a[q_1, Z_1, q_2][q'_2, Z_2, q_3] \dots [q'_n, Z_n, q_{n+1}]$ eine Linksableitung ist, dann gilt $q_2 = q'_2, q_3 = q'_3, \dots, q_n = q'_n$.
Damit läßt sich dann per Induktion zeigen:

$$\begin{array}{ccc} [q, Z, q_{n+1}] & \overset{L}{\Rightarrow}{}^*_G & w[q_1, Z_1, q_2][q_2, Z_2, q_3] \dots [q_n, Z_n, q_{n+1}] \\ & \Updownarrow & \\ qZ & \vdash^*_A & wq_1 Z_1, \dots, Z_n \end{array}$$

Mit den Regeln $S \to [q_0, Z_0, q]$ für alle $q \in Q$ gilt dann:

$$\begin{array}{ccccc} [q_0, Z_0, q_1] & \Rightarrow^*_G & w & \text{gdw. } w \in L(G) \\ & \Updownarrow & & \\ q_0 Z_0 & \vdash^*_A & wq_1 & \text{gdw. } w \in N(A) \end{array}$$

Also gilt $L(G) = N(G)$. □

Die Linksableitung der Grammatik entspricht wieder genau der Berechnung des Kellerautomaten.

Bemerkungen:

1. Die Sätze 3.22 und 3.23 beweisen, dass Kellerautomaten genau die kontextfreien Sprachen akzeptieren können.

2. Achtung: Diese Aussage gilt nicht, wenn wir uns auf deterministische Kellerautomaten beschränken. Es gibt kontextfreie Sprachen, für die es keinen deterministischen Kellerautomaten gibt, der diese Sprache akzeptiert (weder über Endzustände noch mit leerem Keller).

3. Der Beweis von Satz 3.22 zeigt zusammen mit Satz 3.23, dass Kellerautomaten, die eine Sprache mit leerem Keller akzeptieren nur einen Zustand benötigen.

   Die Zustände also bei einem Kellerautomaten mit der Akzeptierung über den leeren Keller eigentlich überflüssig. Für einige Konstruktionen sind jedoch zusätzliche Zustände hilfreich. Wir werden später in der Übung (Blatt 9, Aufgabe 1d.) sehen, daß für Kellerautomaten mit Akzeptierung über Endzustände zwei Zustände genügen.

3 Eigenschaften kontextfreier Sprachen

Ähnlich wie bei regulären Sprachen untersuchen wir nun die Eigenschaften der kontextfreien Sprachen.

3.1 Eigenschaften einer kontextfreien Sprache

Auch für kontextfreie Sprachen gibt es ein Pumping-Lemma. Allerdings ist das Pumping-Lemma für kontextfreie Sprachen etwas komplizierter.

Idee: Wenn eine Grammatik eine Sprache L mit unendlich vielen Wörtern erzeugt, dann muß es Wörter geben, in deren Ableitungsbaum auf einem Pfad dieselbe Variable mehrfach vorkommt.

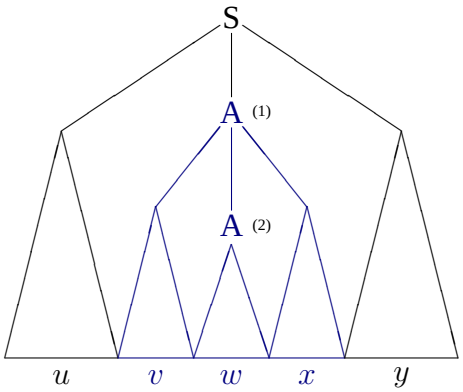


Abbildung 3.9. Ein Ableitungsbaum mit zwei gleichen Variablen A auf einem Pfad.

In Abb. 3.9 ist ein solcher Ableitungsbaum für ein Wort $z = uvwxy$ schematisch dargestellt. Dabei können wir den bei Knoten (1) beginnenden Teilbaum so wählen, daß auf keinem Pfad dieses Teilbaumes zwei gleiche Variable vorkommen. Nun können wir das Teilstück des Baumes zwischen (1) und (2) beliebig häufig wiederholen und erhalten so Ableitungsbäume für die Wörter uv^iwx^iy für jedes $i \in \mathbb{N}$. Da in dem bei (1) beginnenden Teilbaum auf keinem Pfad weitere Variablen doppelt vorkommen gilt $|vwx| \leq n$ für ein geeignetes n, das nur von der Grammatik abhängt. Außerdem können wir davon ausgehen, daß in der Grammatik keine ε-Produktionen vorkommen. Also gilt $|vx| \geq 1$.

> *Die Zahl n ergibt sich aus $k^{|V|+1}$, wobei k die Länge der längste rechte Seite einer Produktion von G ist.*

Eine präzise Formulierung dieser Überlegung führt zum Pumping-Lemma für kontextfreie Sprachen:

Lemma 3.24 (Pumping-Lemma) *Für jede kontextfreie Sprache L gibt es ein $n \in \mathbb{N}$, so dass für jedes Wort $z \in L$ mit $|z| \geq n$ eine Zerlegung $z = uvwxy$ mit $|vx| \geq 1$ und $|vwx| \leq n$ existiert, für die $uv^iwx^iy \in L$ für alle $i \in \mathbb{N}$ gilt.*

Beweis: Entweder: Formalisieren der Vorüberlegung.
Oder: Spezialfall von Ogden's Lemma (Markierung aller Buchstaben), das wir noch beweisen werden. □

Beispiel 3.9 (Anwendung des Pumping-Lemma)
Wir beweisen, daß die Sprache $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ nicht kontextfrei ist. Angenommen L wäre kontextfrei, dann gäbe es ein $n \in \mathbb{N}$, so dass für jedes Wort $z \in L$ mit $|z| \geq n$ eine Zerlegung $z = uvwxy$ mit $|vx| \geq 1$ und $|vwx| \leq n$ existiert, für die $uv^i wx^i y \in L$ für alle $i \in \mathbb{N}$ gilt.
Wir wählen für dieses n das Wort $z = a^n b^n c^n$ und eine beliebige Zerlegung $z = uvwxy$ mit $|vwx| \leq n$ und $|vx| \geq 1$. Wir werden dann zeigen, daß für diese Zerlegung für mind. ein $i \in \mathbb{N}$ nicht $uv^i wx^i y \in L$ gilt:
Sei also $z = uvwxy$ eine beliebige Zerlegung von z mit $|vwx| \leq n$ und $|vx| \geq 1$: Wegen $|vwx| \leq n$ können v und x entweder nur a's und b's oder nur b's und c's enthalten. Wegen $|vx| \geq 1$ muß vx mindestens ein Zeichen enthalten. Dann liegt aber uv^2wx^2y nicht in L, da uv^2wx^2y entweder mehr a's als c's oder mehr c's als a's enthält.

Bemerkung: Wie beim uvw-Theorem für reguläre Sprachen gibt es nicht kontextfreie Sprachen, für die das nicht mit Hilfe von Lemma 3.24 bewiesen werden kann:

$$L = \{a^i b^j c^k d^l \mid i = 0 \text{ oder } j = k = l\}$$

- $b^* c^* d^*$ kann immer aufgepumpt werden

- $a^+ b^n c^n d^n$ kann mit $uvwx = a^+$ und $y = a^* b^n c^n d^n$ auch immer gepumpt werden.

Verallgemeinerung: Wir wollen bestimmte Teile des Wortes z markieren, die gepumpt werden sollen.

Zur Markierung kann man beispielsweise bestimmte Buchstaben des Wortes unterstreichen. Die unterstrichenen Buchstaben nennen wir dann markierte Buchstaben.

Lemma 3.25 (Ogden's Lemma) *Für jede kontextfreie Sprache L gibt es ein $n \in \mathbb{N}$, so dass für jedes $z \in L$ gilt:*

Wenn in z mindestens n Buchstaben markiert sind, dann gibt es eine Zerlegung $z = uvwxy$, so dass mindestens ein Buchstabe von v oder x markiert ist und höchstens n Buchstaben von vwx, markiert sind und $uv^i wx^i y \in L$ für alle $i \in \mathbb{N}$.

Beweis: Hier skizzieren wir nur die Idee: Ohne Beschränkung der Allgemeinheit können wir davon ausgehen, dass L von einer Grammatik $G = (V, \Sigma, P, S)$ in Chomsky-Normalform akzeptiert wird.

Die Spezialfälle, für die es keine Grammatik in Chomsky-Normalform gibt, kann man sich leicht separat überlegen.

Wir wählen $n = 2^{|V|+1}$. Sei nun $z \in L$ ein Wort, in dem mindestens n Buchstaben markiert sind. Wir betrachten einen Ableitungsbaum B von w. Die markierten Buchstaben entsprechen dann markierten Blättern des Baumes. Dieser Baum ist im wesentlichen ein Binärbaum (vergleiche mit den Bemerkungen zu Satz 3.13).

In diesem Baum wählen wir nun ausgehend von der Wurzel einen Pfad von der Wurzel zu einem Blatt aus. Den Pfad wählen wir immer in der Richtung des Unterbaumes, in dem mehr markierte Blätter liegen (wenn in beiden Bäumen gleich viele markierte Blätter liegen, können wir uns eine Alternative aussuchen).

Auf diesem Pfad müssen wegen $n = 2^{|V|+1}$ und der Wahl des Pfades mindestens $|V|+1$ Knoten liegen, für die sowohl der rechte als auch der linke Unterbaum ein markiertes Blatt enthalten[3]. Diese Knoten nennen wir nun *Verzweigungspunkte*.

Auf dem gewählten Pfad müssen also mindestens zwei Verzweigungspunkte mit der selben Variablen beschriftet sein. Wir können aus den letzten $|V| + 1$ Verzweigungspunkten also zwei Knoten v_1 und v_2 auswählen, die mit derselben Variable A beschriftet sind. Dieser Baum ist in Abb. 3.10 dargestellt. Aus diesem Baum ergibt sich eine Zerlegung $z = uvwxy$.

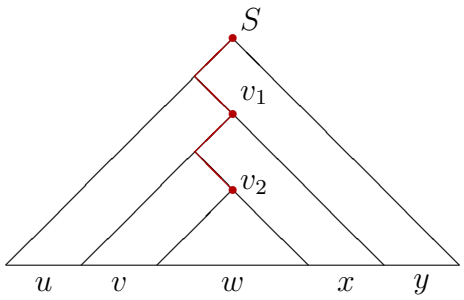


Abbildung 3.10. Ableitungsbaum mit einem Pfad mit zwei gleich beschrifteten Knoten v_1 und v_2.

Da v_1 Verzweigungspunkt ist, enthält entweder v oder x mindestens ein markiertes Blatt. Außerdem enthält vwx höchstens $n = 2^{|V|+1}$ markierte Blätter, da nach v_1 höchstens $|V|$ Verzweigungspunkten vorkommen[4].

Offensichtlich können wir für jedes $i \in \mathbb{N}$ aus dem Ableitungsbaum für $uvwxy$ auch einen Ableitungsbaum für uv^iwx^iy konstruieren (durch Weglassen bzw. wiederholen des Pfades $v_1 \to v_2$). Damit ist mit $uvwxy$ eine Zerlegung von z mit den geforderten Eigenschaften gefunden. □

Bemerkung: Es gibt eine analoge Verallgemeinerung des uvw-Theorems für reguläre Sprachen (Lemma 2.11). Diese Verallgemeinerung heißt der *Satz vom iterierenden Faktor*.

[3] Denn nur an solchen Knoten kann die Anzahl der markierten Blätter des Unterbaum abnehmen, und zwar bestenfalls um die Hälfte.

[4] Ein Blatt enthält höchstens einen markierten Buchstaben. Rückwärts entlang des Pfades kann sich die die Anzahl der markierten Blätter eines Unterbaumes nur an einem Verzweigungspunkt erhöhen – und zwar bestenfalls verdoppeln.

3.2 Abschlußeigenschaften kontextfreier Sprachen

In diesem Abschnitt untersuchen wir die Abschlußeigenschaften der kontextfreien Sprachen. Wir betrachten dieselben Operationen wie für reguläre Sprachen – allerdings sind die kontextfreien Sprachen nicht unter allen Operationen abgeschlossen.
Zunächst die guten Nachrichten:

Satz 3.26 (Abgeschlossenheit kontextfreier Sprachen) *Die kontextfreien Sprachen sind abgeschlossen unter*

Operation	***Beweisidee***
Vereinigung	$S \to S_1 \mid S_2$
Konkatenation	$S \to S_1 S_2$
Kleene'sche Hülle	$S \to \varepsilon \mid S_1 S$
Substitution	$A \to aBCdE \rightsquigarrow A \to S_a BC S_d E$
Homomorphismen	*Spezialfall der Substitution*
Inversen Homomorphismen	*Simulation des Kellerautomaten auf $h(w)$, wobei $h(w)$ zeichenweise berechnet wird und in einen endlichen Puffer zwischengespeicher wird.*
Spiegelung	$A \to x_1 \dots x_n \rightsquigarrow A \to x_n \dots x_1$

Leider gibt es auch ein paar schlechte Nachrichten:

Satz 3.27 *Die kontextfreien Sprachen sind nicht abgeschlossen unter:*

Operation	***Beweisidee***
Durchschnitt	*Gegenbeispiel: Für die kontextfreien Sprachen $L_1 = a^n b^n c^*$ und $L_1 = a^* b^n c^n$ ist der Durchschnitt $L_1 \cap L_2 = a^n b^n c^n$ nicht kontextfrei.*
Komplement	*Wären die kontextfreien Sprachen unter Komplement abgeschlossen, müßten sie auch unter Durchschnitt abgeschlossen sein, da gilt $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.*
Quotientenbildung	*Ohne Beweis. In der Übung (Blatt 9 Aufgabe 3) werden wir aber beweisen, dass kontextfreie Sprachen nicht effektiv unter der Quotientenbildung abgeschlossen sind.*

Kontextfreie Sprachen sind nicht unter Quotientenbildung abgeschlossen. Wenn man allerdings nur den Quotienten mit einer regulären Sprache bildet, ist das Ergebnis wieder kontextfrei:

Satz 3.28 (Durchschnitt und Quotientenbildung mit regulären Sprachen) *Sei R eine reguläre Sprache, dann sind die kontextfreien Sprachen effektiv abgeschlossen unter:*

Operation	***Beweisidee***
Durchschnitt mit einer regulären Sprache $L \cap R$	*Wir lassen den Kellerautomaten für L mit dem endlichen Automaten für R synchron laufen.*
Quotientenbildung mit einer regulären Sprache L/R	*Wird in der Übung (Blatt 9, Aufgabe 3) bewiesen: unter Ausnutzung des Abschlusses unter Homomorphismen, inversen Homomorphismen und dem Durchschnitt mit einer regulären Sprache.*

4 Entscheidbarkeitseigenschaften

Nun wenden wir uns den Entscheidbarkeitseigenschaften für kontextfreie Sprachen zu. Einige Eigenschaften sind auch für kontextfreie Sprachen entscheidbar. Diese werden wir uns in Abschnitt 4.1 ansehen. Allerdings gibt es bereits für kontextfreie Sprachen einige – auf den ersten Blick einfache – Eigenschaften, die für kontextfreie Sprachen nicht entscheidbar sind. Diese werden wir in Abschnitt 4.2 ansehen.

4.1 Entscheidungsverfahren

Satz 3.29 (Leerheit/Endlichkeit) *Für jede kontextfreie Sprache L (repräsentiert durch eine kontextfreie Grammatik oder einen Kellerautomaten) ist entscheidbar, ob*

- *$L = \varnothing$ gilt und ob*
- *$|L| = \omega$*

Beweis: O.B.d.A. gehen wir davon aus, daß die Sprache L durch eine kontextfreie Grammatik G mit $L(G) = L$ repräsentiert ist.
Um die Frage $L = \varnothing$ zu entscheiden, müssen wir entscheiden, ob das Axiom der Grammatik G produktiv ist. Die Menge der produktiven Variablen einer Grammatik ist effektiv bestimmbar. Also ist $L = \varnothing$ entscheidbar: $L = \varnothing$ genau dann, wenn $S \notin V^G_{prod}$.

Um die Frage $L = \omega$ zu entscheiden, nehmen wir o.B.d.A an, dass G in CNF gegeben ist (insbesondere kommen in G keine nutzlosen Variablen und keine Kettenproduktionen vor). Wir konstruieren nun einen Graphen, dessen Knoten die Variablen V der Grammatik sind. Der

Graph enthält eine Kante (A, B) genau dann, wenn die Grammatik eine Produktion $A \to BC$ oder $A \to CB$ enthält. Dieser Graph läßt sich einfach berechnen.
Dann gilt $L = \omega$ genau dann, wenn der Graph einen Zyklus (Kreis) besitzt. Da entscheidbar ist, ob ein Graph einen Kreis besitzt, ist damit auch $L = \omega$ entscheidbar. □

Satz 3.30 (Wortproblem) *Für ein Wort $w \in \Sigma^*$ und eine kontextfreie Sprache L ist entscheidbar, ob $w \in L$ gilt.*

Beweis: Wir haben bereits bei der Chomsky- und der Greibach-Normalform gesehen, daß das Wortproblem für kontextfreie Sprachen entscheidbar ist, da wir nur endlich (wenn auch sehr viele) Ableitungen ansehen müssen. □

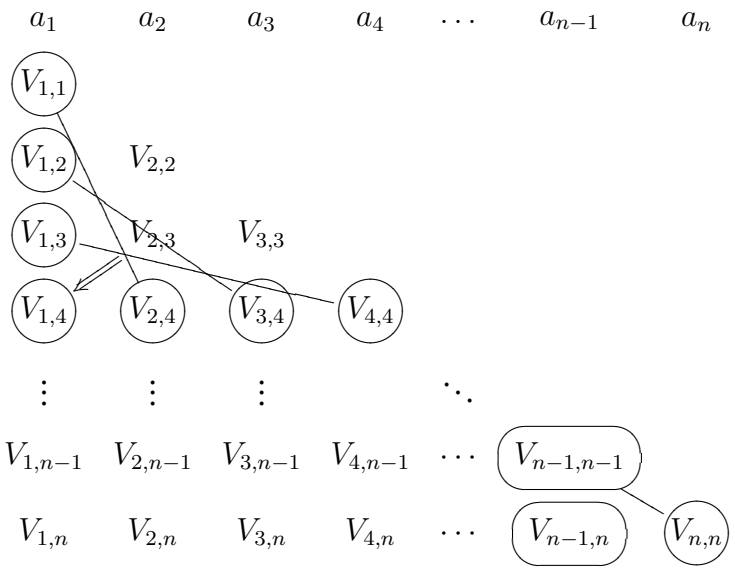
Bemerkung: Das Durchprobieren aller Ableitungen einer bestimmten Länge ist ein extrem ineffizientes Entscheidungsverfahren für das Wortproblem. Mit Hilfe des Algorithmus von Cocke, Younger und Kasami (*CYK-Algorithmus*), kann man viel effizienter entscheiden, ob ein Wort w in einer kontextfreien Sprache L liegt. Wir gehen dabei davon aus, daß die Sprache durch eine Grammatik in Chomsky-Normalform gegeben ist (sie ist ggf. effektiv konstruierbar).
Idee (des CYK-Algorithmus): Für das Wort $w = a_1 \dots a_n$ bestimmt man für alle $j \geq i \geq 1$ die folgenden Mengen:

$$V_{i,j} = \{A \in V \mid A \Rightarrow_G^* a_i \dots a_j\}$$

Offensichtlich gilt $w \in L$ genau dann, wenn $S \in V_{1,n}$.
Die $V_{i,j}$ können wir (für eine Grammatik in CNF) systematisch bestimmen:

- $V_{i,i} = \{A \in V \mid A \to a_i \in P\}$
- $V_{i,j} = \bigcup_{k=i}^{j-1} \{A \in V \mid A \to BC \in P, B \in V_{i,k}, C \in V_{k+1,j}\}$

Die $V_{i,j}$ lassen sich systematisch gemäß des folgenden Schemas ausgehend von der Diagonale bestimmen (Prinzip der dynamischen Programmierung):



Achtung: Dieses Schema unterscheidet sich etwas von dem Schema, dass in Informatik IV vorgestellt wurde.

Dieses Schema kann man derart erweitern, daß nicht nur entschieden wird, ob ein Wort zur erzeugten Sprache der Grammatik gehört, sondern auch der entsprechende Ableitungsbaum erzeugt wird.

Beispiel 3.10
Wir überprüfen mit Hilfe des CYK-Algorithmus, ob das Wort $w = abbb$ von der folgenden Grammatik G erzeugt wird:

$S \to AA$
$A \to CS \mid SS$
$A \to b$
$C \to a$

$w =$	a	b	b	b
	$\{C\}$			
	$\varnothing$	$\{A\}$		
	$\{A\}$	$\{S\}$	$\{A\}$	
	$\{S\}$	$\varnothing$	$\{S\}$	$\{A\}$

Wegen $S \in V_{1,4} = \{S\}$ gilt $w \in L(G)$.

In dem Beispiel haben die Mengen $V_{i,j}$ höchstens ein Element; das liegt aber an der speziellen Struktur des Beispiels; im allgemeinen können die Mengen $V_{i,j}$ mehr als ein Element enthalten.

4.2 Unentscheidbarkeitsresultate

Viele Probleme, die für reguläre Sprachen entscheidbar sind, sind für kontextfreie Sprachen breits unentscheidbar.

Satz 3.31 (Unentscheidbare Eigenschaften kontextfreier Sprachen)
Die folgenden Eigenschaften für kontextfreie Sprachen sind nicht entscheidbar:

- $L(G) = \Sigma^*$
- $L(G_1) = L(G_2)$ *(Äquivalenz)*
- $L(G_1) \subseteq L(G_2)$ *(Inklusion)*
- $L(G_1) \cap L(G_2) = \varnothing$ *(Disjunktheit)*
- $L(G_1) \cap L(G_2)$ *kontextfrei ?*
- $L(G)$ *regulär ?*
- $\overline{L(G)}$ *kontextfrei ?*

Beweis: Der Beweis ist uns im Moment noch nicht möglich. Das wesentliche Argument für die Unentscheidbarkeit der Disjunktheit ist, dass wir die “gültigen Berechnungen” einer Turing-Maschine als Durchschnitt zweier kontextfreier Grammatiken formulieren können. Wir kommen später darauf zurück (siehe Satz 5.12 und Übungsblatt 11, Aufgabe 2 und Übungsblatt 12, Aufgabe 1.c). □

5 Deterministisch kontextfreie Sprachen

5.1 Motivation und Überblick

In Abschnitt 2 haben wir ein aus theoretischer Sicht sehr schönes Resultat kennengelernt: Die kontextfreien Sprachen sind genau die Sprachen, die von nicht-deterministischen Kellerautomaten akzeptiert werden. Der Preis für diese Äquivalenz ist der Nicht-Determinismus (im Gegensatz zu endlichen Automaten können wir nicht jeden nicht-deterministischen Kellerautomaten in einen äquivalenten deterministischen Kellerautomaten überführen).
Für die Konstruktion von Parsern sind nicht-deterministische Kellerautomaten nicht besonders geeignet, da der Nicht-determinismus irgendwie simuliert werden muß (z.B. durch Backtracking) das ist aber sehr ineffizient. Selbst wenn man den effizienteren CYK-Algorithmus einsetzt, benötigt das Parsen $O(n^3)$ Zeit in der Länge der Eingabe n. Dies ist für die meisten praktische Anwendung (Compilerbau) viel zu aufwendig[5].
Wenn der Kellerautomat deterministisch ist, können wir ein Wort in linearer Zeit (d.h. $O(n)$) parsen. Das ist das, was wir uns für die Praxis wünschen. Deshalb betrachten wir diese Sprachen etwas genauer; noch ausführlich werden sie natürlich in der Vorlesung Compilerbau betrachtet.

Definition 3.32 (Deterministisch kontextfreie Sprache) *Eine kontextfreie Sprache L heißt deterministisch, wenn es einen deterministischen Kellerautomaten A gibt, der die Sprache L über Endzustände akzeptiert (d.h. $L = L(A)$).*

Bemerkungen:

1. Wenn wir die in Definition 3.32 Akzeptierung mit leeren Keller wählen, ist die Definition der deterministisch kontextfreien Sprachen echt schwächer!

   Eine von einem deterministischen Kellerautomat mit leerem Keller akzeptierte Sprache ist präfixfrei. Dies gilt für Sprachen die von einem deterministischen Kellerautomaten über Endzustände akzeptiert wird nicht.

2. Es gibt kontextfreie Sprachen, die nicht deterministisch sind! Beispielsweise ist die Sprache $\{w\overleftarrow{w} \mid w \in \Sigma^*\}$ für $|\Sigma| \geq 2$ kontextfrei aber nicht deterministisch! Wir verzichten hier auf einen Beweis und geben lediglich eine Plausibilitätserklärung: Der Kellerautomat muß die Mitte des eingelesenen Wortes erraten.

   Die Klasse der deterministisch kontextfreien Sprachen liegt also echt zwischen der Klasse der regulären Sprachen und der Klasse der kontextfreien Sprachen und „neben“ den linearen Sprachen.

Bevor wir uns der Syntaxanalyse zuwenden untersuchen wir zunächst die Abschlußeigenschaften und die Entscheidbarkeitseigenschaften der deterministisch kontextfreien Sprachen. Da gibt es – im ersten Augenblick vielleicht überraschend – deutliche Unterschiede zu den kontextfreien Sprachen.

[5] Wer mag bei einem vier mal längeren Programm schon 64 mal länger warten, bis es übersetzt ist (1 Min. → 1 Std.).

5.1.1 Abschlußeigenschaften

Deterministisch kontextfreie Sprachen sind abgeschlossen unter:

- Komplementbildung: Die Beweisidee hierfür ist ähnlich wie bei den endlichen Automaten. Der Beweis ist aber wegen der ε-Übergänge etwas komplizierter.

Deterministisch kontextfreie Sprachen sind *nicht* abgeschlossen unter:

- Durchschnittsbildung: Der Beweis geht wie für kontextfreien Sprachen: $a^n b^n c^*$ und $a^* b^n c^n$ sind deterministisch kontextfrei; ihr Durchschnitt $a^n b^n c^n$ ist nicht einmal kontextfrei.

- Vereinigung: Sonst wären sie mit dem Abschluß unter Komplement und dem Gesetz von de Morgan auch unter Durchschnitt abgeschlossen.

- Konkatenation: (ohne Beweis)

- Kleen'sche Hüllenbildung: (ohne Beweis)

Wie sieht es mit dem Abschluß der deterministisch kontextfreien Sprachen unter Homomorphismen und inversen Homomorphismen aus?
Unter den Homomorphismen ist die Klasse der deterministisch kontextfreie Sprachen nicht abgeschlossen (Beispiel: Testat 2).
Unter inversen Homomorphismus hingegen schon. Das liegt daran, daß man den Abschluß unter inversen Homomorphismen über das Automatenmodell der Sprachklasse zeigt. Und deterministisch kontextfreie Sprachen sind (nur) über das Automatenmodell definiert.

5.1.2 Entscheidbarkeit

Einige Probleme, die für kontextfreie Sprachen nicht entscheidbar sind, sind für deterministisch kontextfreie Sprachen entscheidbar.
Für eine deterministisch kontextfreie Sprache L ist (wenn sie als deterministischer Kellerautomat gegeben ist) entscheidbar, ob

- $L = \Sigma^*$ gilt? Denn wegen des Abschlusses unter Komplement ist $\overline{L}$ deterministisch kontextfrei und insbesondere kontextfrei. Es gilt $L = \Sigma^*$ gdw. $\overline{L} = \varnothing$; diese Frage ist wegen Lemma 3.29 entscheidbar.

- $\overline{L}$ deterministisch kontextfrei ist? Dies ist trivial ("ja"), da die deterministisch kontextfreien Sprachen unter Komplementbildung abgeschlossen sind.

5.2 Syntaxanalyse

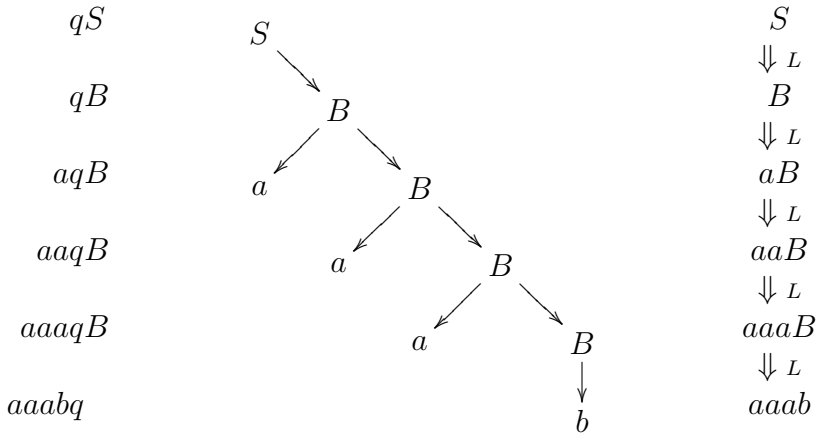
In diesem Abschnitt werden wir die Prinzipien der Top-down- und Bottom-up-Syntaxanalyse anhand von Beispielen verdeutlichen. Hier geht es nur darum, ein „Gefühl“ für diese Prinzipien zu entwickeln. Für eine ausführlichere und formalere Darstellung verweisen wir auf die Literatur zum Compilerbau.

Um die Unterschiede zwischen Top-down- und Bottom-up-Analyse zu Verdeutlichen benutzen wir ein laufendes Beispiel. Unsere Beispielsprache ist regulär; wir schießen also eigentlich mit Kanonen auf Spatzen. In der Praxis würde man weder Bottom-up noch Top-down-Analyse betreiben, sondern einen endlichen Automaten bemühen. Trotzdem läßt das Beispiel die wesentlichen Ideen ganz gut erkennen.

Beispiel 3.11 (Top-down-Analyse)
Wir betrachten die folgende Grammatik, die die reguläre Sprache $a^*b + a^*c$ erzeugt:

$$\begin{array}{lcl} S & \to & B \mid C \\ B & \to & aB \mid b \\ C & \to & aC \mid c \end{array}$$

Nun betrachten wir die Berechnung eines Kellerautomaten, der diese Sprache akzeptiert. Der Automat wird wie im Beweis von Satz 3.22 konstruiert – wir verzichten hier darauf, den Automaten explizit anzugeben. Die Berechnung des Automaten für das Wort $aaab$ ist auf der linken Seite dargestellt:



In der Mitte ist der zu dieser Berechnung gehörige Ableitungsbaum dargestellt und auf der rechten Seite die zugehörige Linksableitung.

In diesem Beispiel wird im Laufe der Berechnung der Ableitungsbaum von oben nach unten aufgebaut. Dabei wir während der Berechnung das Eingabewort von links nach recht eingelesen; die Berechnung entspricht einer *L*inksableitung der Grammatik. Deshalb spricht man bei diesem Vorgehen vom *Top-down-Prinzip* bzw. *LL-Prinzip*.

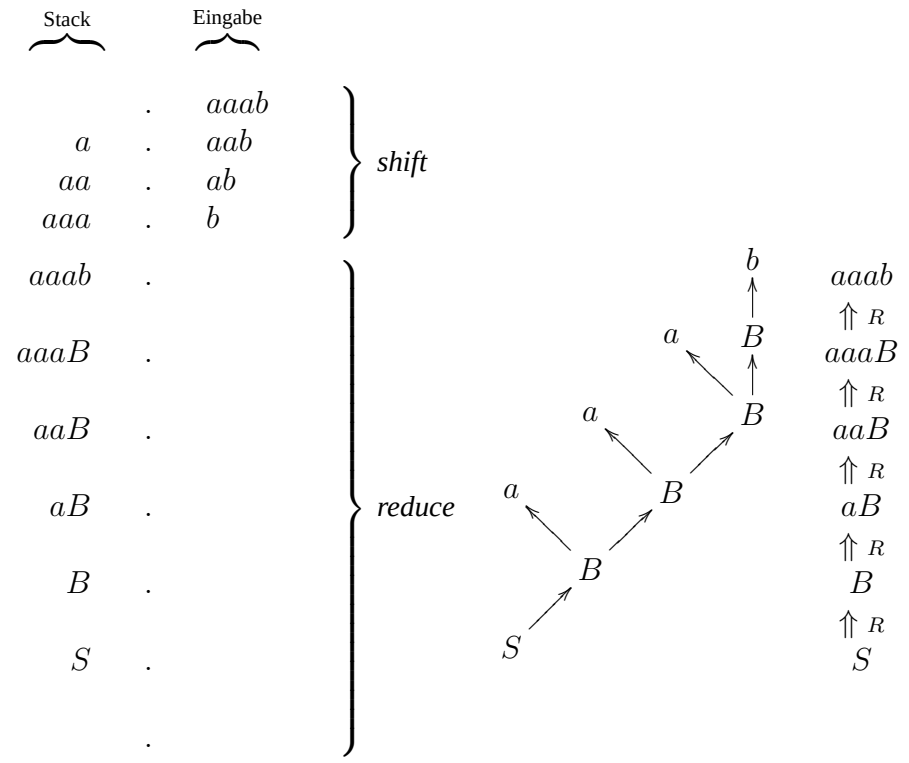
> *Das erste* L *in* LL *steht für die Leserichtung (von links nach rechts). Das zweite* L *steht für die Berechnung einer* L*inksableitung.*

Im Beispiel haben wir am Anfang der Berechnung die Produktion $S \to B$ ausgewählt, weil nur so später das b am Ende des Wortes erzeugt werden kann. Um die richtige Entscheidung zu treffen, muß man also bereits beim Lesen des ersten Zeichens das letzte Zeichen der Eingabe kennen. Man kann die richtige Entscheidung also nicht durch beschränkte *Vorausschau* (lookahead) von k Zeichen treffen. Der Kellerautomat ist deshalb nicht deterministisch und die Grammatik ist keine *LL(k)-Grammatik*. Und der Kellerautomat ist kein *LL(k)-Parser*. Wenn sich dagegen in allen Situationen die richtige Entscheidung aufgrund der nächsten k Eingabezeichen treffen läßt, spricht man von einer LL(k)-Grammatik bzw. einem LL(k)-Parser.

Wir können nun versuchen, den Ableitungsbaum von unten nach oben aufzubauen.

Beispiel 3.12
Hierzu verwenden wir wieder die Grammatik aus Beispiel 3.11 und geben eine Berechnung eines entsprechenden Kellerautomaten an. Allerdings stellen wir die Konfigurationen der Berechung jetzt etwas anders dar. Jetzt steht der Keller auf der linken Seite und die restliche Eingabe

– getrennt durch einen Punkt – auf der rechten Seite: $\alpha.w$, wobei α der aktuelle Kellerinhalt und w die restliche Eingabe ist.
Der Kellerautomat schiebt die Eingabezeichen auf seinen Keller (shift-Aktion) und versucht dann, die obersten Zeichen auf seinem Keller mit der rechten Seite einer Regel der Grammatik zu identifizieren und dann durch die linke Seite dieser Regel zu ersetzen (reduce-Aktion). Die Berechung ist auf der linken Seite dargestellt. In der Mitte ist der Ableitungsbaum dargestellt; da er von unten nach oben erzeugt wird, ist er hier auf dem Kopf dargestellt. Rechts ist die entsprechende Rechtsableitung dargestellt:



Im Beispiel werden zunächst nur shift-Aktionen ausgeführt und dann nur reduce-Aktionen. Im allgemeinen treten diese Aktionen jdoch beliebig gemischt auf.

Da der Ableitungsbaum von unten nach oben aufgebaut wird, nennen wir einen Kellerautomaten, der nach diesem Prinzip vorgeht, einen *Bottom-up-Parser*. Da die Berechnung einer rückwärts gelesenen Rechtsableitung entspricht, wird er auch *LR-Parser* genannt[6].
In unserem Beispiel ist die Konstruktion des Ableitungsbaumes von unten nach oben deterministisch – und zwar für alle Eingabewörter. Der Grund dafür ist, daß im Beispiel erst die gesamte Eingabe auf den Keller geschoben wird; das letzte Eingabezeichen entscheidet dann, ob wir zu B oder zu C reduzieren. Ganz allgemein gilt:

Merksatz: (deterministische) LR-Parser sind leistungsfähiger als (deterministische) LL-Parser. Eine Plausibilitätserklärung dafür ist, dass LR-Parser (im Extremfall wie in unserem Beispiel) die gesamte Eingabe "ansehen" können, indem sie sie auf den Keller schieben (shift), bevor sie die Ableitung durch Reduktion des Kellers (reduce) generieren.

[6] Das L steht weiterhin für das Lesen der Eingabe von links nach rechts; das R steht für die Konstruktion der Rechtsableitung.

5.3 LR(0)-Parser

Da LR-Parser leistungsfähiger als LL-Parser sind, werden in der Praxis hauptsächlich LR-Parser oder Varianten davon (SLR- oder LALR-Parser) eingesetzt. In diesem Abschnitt stellen wir die wesentlichen Konzepte, die bei der Konstruktion von LR-Parsern eine Rolle spielen, vor. Der Einfachheit halber beschränken wir uns jedoch auf LR(0)-Parser, d.h. Parser die ohne Zeichenvorausschau auskommen. Das ist (bei LR-Parsern) keine wesentliche Einschränkung und die wesentlichen Ideen werden bereits bei der Konstruktion von LR(0)-Parsern deutlich.
Auch hier geht es vor allem darum, ein Gefühl für die Konstruktionen zu entwickel; die theoretischen Grundlagen werden nur am Rande erwähnt.

Bemerkung: Im folgenden gehen wir immer davon aus, dass die betrachteten Grammatiken keine nutzlosen Symbole enthalten. Außerdem soll das Startsymbol in keiner rechten Seite vorkommen (notfalls fügen wir ein neues Startsymbol S' und eine neue Regel $S' \to S$ hinzu).

Idee: Zunächst diskutieren wir die Idee, die der Definition der LR(0)-Grammatiken zugrundeliegt: Angenommen wir haben in einem LR-Parser die Konfiguration $\alpha\beta.u$ und $X \to \beta$ ist eine Regel der Grammatik. Dann könnte der LR-Parser die Reduktion zu $\alpha X.u$ durchführen. Damit der Parser ohne Zeichenvorausschau deterministsisch ist müssen wir gewährleisten, daß es für jede andere (oder sogar dieselbe) restliche Eingabe v keine anderen Aktion gibt, d.h. $\alpha\beta.v$ sollte auch zu $\alpha X.v$ reduziert werden. Dies können wir nun mit Hilfe der Rechtsableitungen der Grammatik etwa wie folgt formulieren:

$$\left.\begin{array}{l} \alpha X u \overset{R}{\Rightarrow} \alpha\beta u \\ \gamma Y w \overset{R}{\Rightarrow} \alpha\beta v \end{array}\right\} \Rightarrow \begin{array}{l} \gamma = \alpha \\ Y = X \\ v = w \end{array} \qquad \begin{array}{l} \alpha, \beta, \gamma \in (V \cup \Sigma)^*, \\ u, v, w \in \Sigma^*, \\ X, Y \in V \end{array}$$

Dabei wird β *Schlüssel* von $\alpha\beta u$ genannt. In der formalen Defintion müssen wir noch fordern, daß $\alpha\beta u$ und $\alpha\beta v$ auch Satzformen der Grammatik sind.

Definition 3.33 (LR(0)-Grammatik, LR(0)-Sprache) *Eine kontextfreie Grammatik G heißt* LR(0)-Grammatik*, wenn für alle $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$ und $u, v, w \in \Sigma^*$ mit*

$$S \overset{R}{\Rightarrow}{}^*_G \alpha X u \overset{R}{\Rightarrow}_G \alpha\beta u$$

$$S \overset{R}{\Rightarrow}{}^*_G \gamma Y w \overset{R}{\Rightarrow}_G \alpha\beta v$$

gilt $\gamma = \alpha$, $Y = X$ und $v = w$.
Eine Sprache heißt LR(0)-Sprache*, wenn es eine LR(0)-Grammatik G gibt, die sie erzeugt.*

Beispiel 3.13
Unsere Beispielgrammatik aus Beispiel 3.11 die im folgenden noch einmal angegeben ist, ist eine LR(0)-Grammatik.

$$\begin{array}{rcl} S & \to & B \mid C \\ B & \to & aB \mid b \\ B & \to & aC \mid c \end{array}$$

Die *Rechtssformen* der Grammatik, d.h. die Satzformen γ mit $S \overset{R}{\Rightarrow}^*_G \gamma$, haben die Form S, a^*B, a^*C, a^*b oder a^*b. Zum Beweis, daß diese Grammatik eine LR(0)-Grammatik ist, müssten wir jetzt für alle diese Satzformen die Definition überprüfen.

Wir betrachten hier exemplarisch den Fall a^*B. Gelte also $S \overset{R}{\Rightarrow}^* \alpha X u \overset{R}{\Rightarrow} \alpha\beta$ mit $\alpha\beta u = a^i B$ für ein $i \in \mathbb{N}$. Wegen $u \in \Sigma^*$ gilt dann $u = \varepsilon$. Wir unterscheiden nun zwei Fälle

$i \geq 1$: Dann gilt $\alpha = a^{i-1}$, $\beta = aB$ und $X = B$.

Sei nun $\gamma Y v$ eine Rechtssatzform mit $\gamma Y v \overset{R}{\Rightarrow} a^i B v$. Dann gilt $v = \varepsilon$, $Y = B$ und $\gamma = a^{i-1} = \alpha$.

$i = 0$: Also $\alpha = \varepsilon$, $\beta = B$ und $X = S$.

Für jede Rechtsatzform $\gamma Y v$ mit $\gamma Y v \overset{R}{\Rightarrow} Bv$ gilt dann ebensfalls $v = \varepsilon$, $\gamma = \varepsilon = \alpha$ und $Y = S$.

Der Beweis für die anderen Fälle ist im wesentlichen analog.

Das explizite Nachrechnen der Defintion ist sehr aufwendig; deshalb werden wir im folgenden eine Technik entwickeln, mit der wir die LR(0)-Eigenschaft einer Grammatik automatisch überprüfen können.

Nun betrachten wir ein Gegenbeispiel

Beispiel 3.14
Die Grammatik

$$\begin{array}{lcl} S & \rightarrow & Bb \mid Cc \\ B & \rightarrow & Ba \mid \varepsilon \\ B & \rightarrow & Ca \mid \varepsilon \end{array}$$

erzeugt dieselbe Sprache wie die Grammatik aus Beispiel 3.13, sie ist jedoch keine LR(0)-Grammatik. Hierzu lässt sich das folgende Gegenbeispiel angeben:

$$S \overset{R}{\Rightarrow}^* Bb \overset{R}{\Rightarrow} b$$

$$S \overset{R}{\Rightarrow}^* Cc \overset{R}{\Rightarrow} c$$

mit $\alpha = \gamma = \varepsilon$, $\beta = \varepsilon$, $u = b$,$v = c$ und $X = B$ und $Y = C$. D.h. wir müssen, ohne die nächste Eingabe (b bzw. c) zu kennen, raten, ob wir $\beta = \varepsilon$ zu B oder zu C reduzieren sollen.

Achtung: *Die von der Grammatik erzeugte Sprache ist eine LR(0)-Sprache, obwohl die Grammatik keine LR(0)-Grammatik ist. Tatsächlich ist die Sprache sogar regulär.*

Wir gehen nun den folgenden Fragen nach:

- Ist entscheidbar, ob eine Grammatik eine LR(0)-Grammatik ist?
- Wenn ja, wie?

Dazu definieren wir einige Begriffe und stellen wir einige Beobachtungen an:

- γ heißt *lebensfähiges Präfix* von G, wenn $\alpha, \beta \in (V \cup \Sigma)^*$, $u \in \Sigma^*$ und $X \in V$ existieren, so daß γ ein Präfix von $\alpha\beta$ ist und gilt: $S \overset{R}{\Rightarrow}^*_G \alpha X u \overset{R}{\Rightarrow}_G \alpha\beta u$.

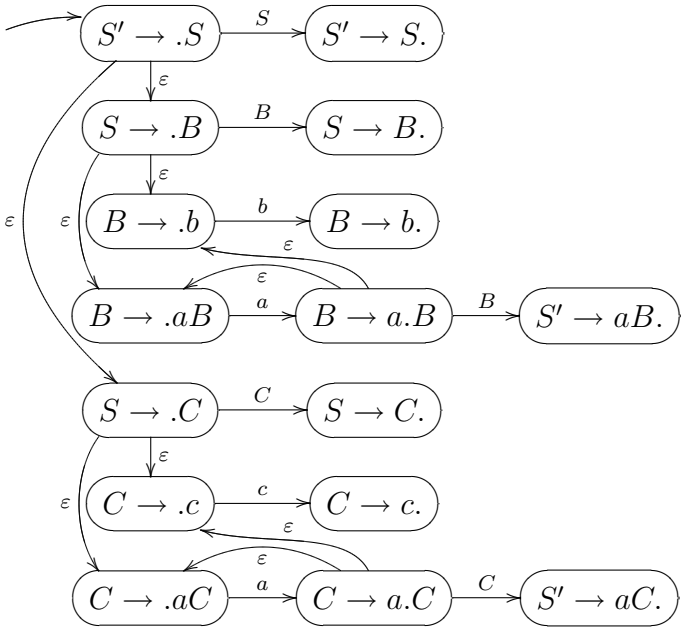
- Die lebensfähigen Präfixe von G entsprechen im wesentlichen den möglichen Kellerinhalten eines LR(0)-Parsers.

- Die lebensfähigen Präfixe von G bilden eine reguläre Sprache.

  Als Beispiel greifen wir wieder auf die Grammatik aus Beispiel 3.11 zurück, fügen aber aus technischen Gründen eine neues Axiom S' hinzu:

$$\begin{array}{lcl} S' & \to & S \\ S & \to & B \mid C \\ B & \to & aB \mid b \\ C & \to & aC \mid c \end{array}$$

  Für diese Grammatik wird die Sprache ihrer lebensfähigen Präfixe vom folgenden endlichen Automaten erkannt (wobei jeder Zustand des Automaten ein Endzustand ist):



  Dieser Automat wird wie folgt konstruiert:

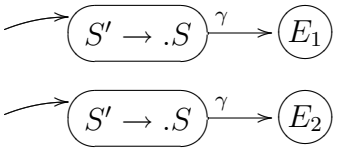
- Für $A \to X_1 \dots X_n$ heißt $A \to X_1 \dots X_i \,.\, X_{i+1} \dots X_n$ ein *Element* von G; die Elemente von G sind die Zustände des Automaten für die lebensfähige Präfixe von G. Ein Element $A \to X_1 \dots X_n.$ heißt *vollständig*; in einem solchen Zustand kann reduziert werden.

- Von jedem Element $A \to X_1 \dots X_i \,.\, X_{i+1} \dots X_n$ gibt es einen Übergang mit X_{i+1} zu dem Element $A \to X_1 \dots X_{i+1} \,.\, X_{i+2} \dots X_n$.

  Von jedem Element $A \to X_1 \dots X_i \,.\, X_{i+1} \dots X_n$ mit $X_{i+1} = A \in V$ gibt es einen ε-Übergang zu jedem Element der Form $A \to .Y_1 \dots Y_k$.

- Wir sagen zwei *Elemente stehen in Konflikt*, wenn entweder beide vollstänig sind (reduce/reduce-Konflikt) oder wenn ein Element vollständig ist und das andere von der Form $A \to X_1 \dots X_i.aX_{i+1} \dots X_n$ (shift/reduce-Konflikt).

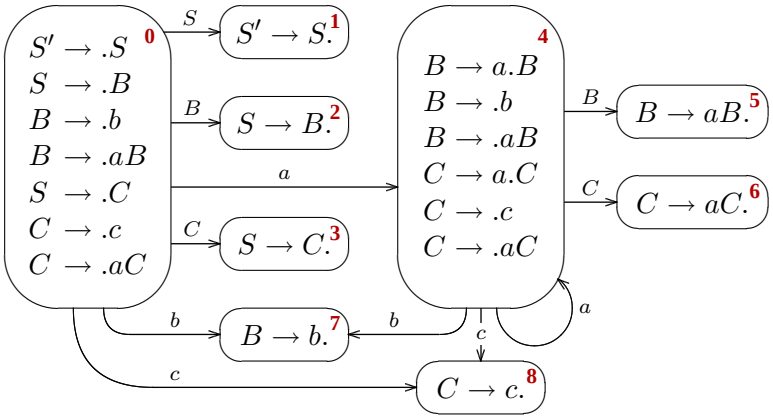
- Die Grammatik ist eine LR(0)-Grammatik, wenn für kein Wort $\gamma \in (V \cup \Sigma)^*$ gilt:



und E_1 und E_2 sind zwei verschiedene Elemente, die in Konflikt stehen.

D.h. für kein lebensfähiges Präfix γ von G besteht ein shift/reduce- oder ein reduce/reduce-Konflikt.

- Diese Bedingung können wir einfach überprüfen, indem wir den zugehörigen deterministischen Automaten konstruieren:



Wenn in diesem deterministischen Automaten in irgendeinem Zustand zwei Elemente vorkommen, die in Konflikt stehen, dann ist die Grammatik nicht LR(0); andernfalls ist sie LR(0).

Zusammenfassung: Entscheidungsverfahren für die LR(0)-Eigenschaft einer Grammatik G:

- Konstruiere den endlichen Automaten für die lebensfähigen Präfixe von G; dieser Automat ist im allgemeinen wegen der ε-Übergänge nicht deterministisch.
- Konstruiere den zugehörigen deterministischen endlichen Automaten.
- Überprüfe, ob in einem Zustand zwei Elemente vorkommen, die in Konflikt stehen.
  - Wenn ja, dann ist G keine LR(0)-Grammatik!
  - Wenn nein, dann ist G eine LR(0)-Grammatik!

Aus dem deterministischen Automaten für die lebensfähigen Präfixe läßt sich unmittelbar ein LR-Parser für die Grammatik konstruieren. Mit Hilfe des endlichen Automaten wird dabei entschieden, ob als nächstes eine shift-Aktion oder eine reduce-Aktion ausgeführt werden soll. Die Idee werden wir hier wieder nur anhand eines Beispiels vorstellen (etwas genauer werden wir uns die Konstruktion in der Übung ansehen: Blatt 10, Aufgabe 2).

Beispiel 3.15 (Konstruktion eines LR(0)-Parsers)
Wir verwenden wieder die Grammatik aus Beispiel 3.11. Die nachfolgende Abbildung zeigt, wie der Parser auf dem Wort $aaab$ arbeitet. Dabei werden im Keller neben den lebensfähigen Präfixen auch die Zustände des Automaten gespeichert. Der eigentliche Kellerinhalt und die Zustände des Automaten alternieren. Die rot dargestellten Zahlen für die Zustände des deterministischen endlichen Automaten, die oberste Zahl entspricht dem aktuellen Zustand des deterministischen endlichen Automaten. Wenn auf S' reduziert wird, wird der Keller gelöscht:

$$
\begin{aligned}
&0.aaab\\
&0a4.aab\\
&0a4a4.ab\\
&0a4a4a4.b\\
&0a4a4a4\underline{b7}.\\
&\quad\downarrow\\
&0a4a4\underline{a4B5}.\\
&\quad\downarrow\\
&0a4\underline{a4B5}.\\
&\quad\downarrow\\
&0\underline{a4B5}.\\
&\quad\downarrow\\
&0\underline{B2}.\\
&\quad\downarrow\\
&\underline{0S1}.\\
&\quad\downarrow\\
&\quad .
\end{aligned}
$$

Da in einem Reduktionsschritt des LR-Parsers mehrere (mind. zwei) Kellersymbole verändert werden, muß ein Reduktionsschritt des Parsers durch mehrere Schritte des Kellerautomaten simuliert werden:

- Löschen der rechten Seite vom Keller (nebst den Zuständen dazwischen).
- Berechnung des neuen Zustandes des DEA aus dem nun obersten Zustand des Kellers.
- Schreiben der linken Seite und des neuen Zustandes auf den Keller

Dies wird in der Übung (Blatt 10, Aufgabe 2) noch etwas genauer behandelt.

Bemerkung::

- Jede LR(0)-Grammatik ist eindeutig.
- Die LR(0)-Sprachen sind *genau* die präfixfreien[7] deterministisch kontextfreien Sprachen.

  Wenn L deterministisch kontextfrei ist, ist $L\$$ präfixfrei und deterministisch ($\$$ ist dabei ein eine Sondersymbol zur Markierung des Endes eines Wortes, das sonst nirgends vorkommt). Durch Hinzufügen der Endemarkierung $\$$ können wir für jede deterministisch kontextfreie Sprache einen LR(0)-Parser bauen.

[7] Eine Sprache L heißt *präfixfrei*, wenn für kein Paar $u \sqsubseteq v$ mit $u \neq v$ sowohl $u \in L$ als auch $v \in L$ gilt.

- In der Praxis werden Parser mit k Zeichen Vorausschau benutzt LR(k)-Parser. Die Idee der Definition der LR(k)-Grammatiken ist fast dieselbe wie bei LR(0)-Grammatiken. Allerdings schränken wird die Bedingungen auf Situationen ein, in denen u und v auf den ersten k Zeichen (der Vorausschau) übereinstimmen: Also, wenn u und v auf den ersten k Zeichen übereinstimmen und wenn

$$S \overset{R}{\Rightarrow}{}^*_G \alpha X u \overset{R}{\Rightarrow}_G \alpha\beta u$$
$$S \overset{R}{\Rightarrow}{}^*_G \gamma Y w \overset{R}{\Rightarrow}_G \alpha\beta v$$

  gilt, dann gilt $\gamma = \alpha$, $Y = X$ und $v = w$.

- Bereits für $k = 1$ ist die Klasse der LR(k)-Sprachen genau die Klasse der deterministisch kontextfreien Sprachen. Es gilt also (für die Sprachklasse)

$$\text{LR}(1) \;=\; \text{LR}(2) \;=\; \text{LR}(3) \;=\; \ldots$$

  Allerdings gibt es für jedes $k \in \mathbb{N}$ eine Grammatik die zwar LR($k+1$)-Grammatik ist, aber nicht LR(k)-Grammatik.

- Der Automat für die lebensfähigen Präfixe (unter Berücksichtigung der nächsten k Eingabezeichen) wird ähnlich wie für LR(0)-Grammatiken definiert; auch der zugehörige LR(k)-Parser wird dann analog konstruiert. Die Konstruktion ist aber komplizierter. In der Praxis werden deshalb andere Varianten (SLR(k)- oder LALR(k)-Grammatiken benutzt). Näheres hierzu wird in der Vorlesung „Compilerbau“ behandelt.

6 Zusammenfassung und Überblick

In diesem Abschnitt geben wir noch einmaleinen Überblick über die Hierarchie der von uns in den vorangegangenen Kapiteln betrachteten Sprachklassen.
Die Abbildungen 3.11 und 3.12 zeigen zwei verschiedene Darstellungen der Inklusionsbeziehungen zwischen den verschiedenen Sprachklassen. In Abbildung 3.11 sind zusätzlich Beispiele für Sprachen in den entsprechenden Klassen angegeben; diese Beispiele zeigen insbesondere, daß die linearen Sprachen und die deterministisch kontextfreien Sprachen (bzgl. Mengeninklusion) unvergleichbar sind.
In Abbildung 3.12 haben wir zusätzlich die Klasse der präfixfreien regulären Sprachen aufgenommen, damit die Sprachhierarchie unten etwas „hübscher“ endet (sonst hätten wir zwei unvergleichbare minimale Elemente).

Achtung: Wir müssen immer sorgfältig unterscheiden, ob wir über Sprachklassen oder Grammatiken reden:

- Als Sprachklassen betrachtet gilt $\text{LR}(1) \;=\; \text{LR}(2) \;=\; \text{LR}(3) \;=\; \ldots$

- Aber es gibt eine LR(2)-Grammatik, die keine LR(1)-Grammatik ist.

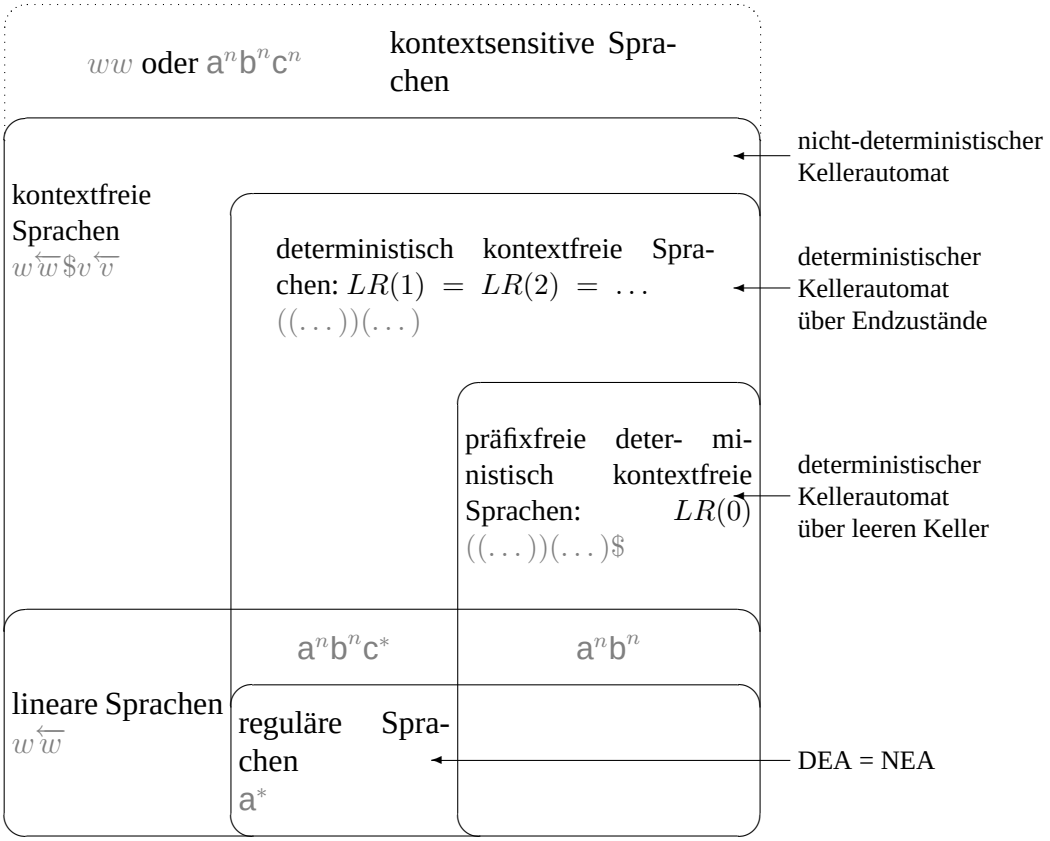
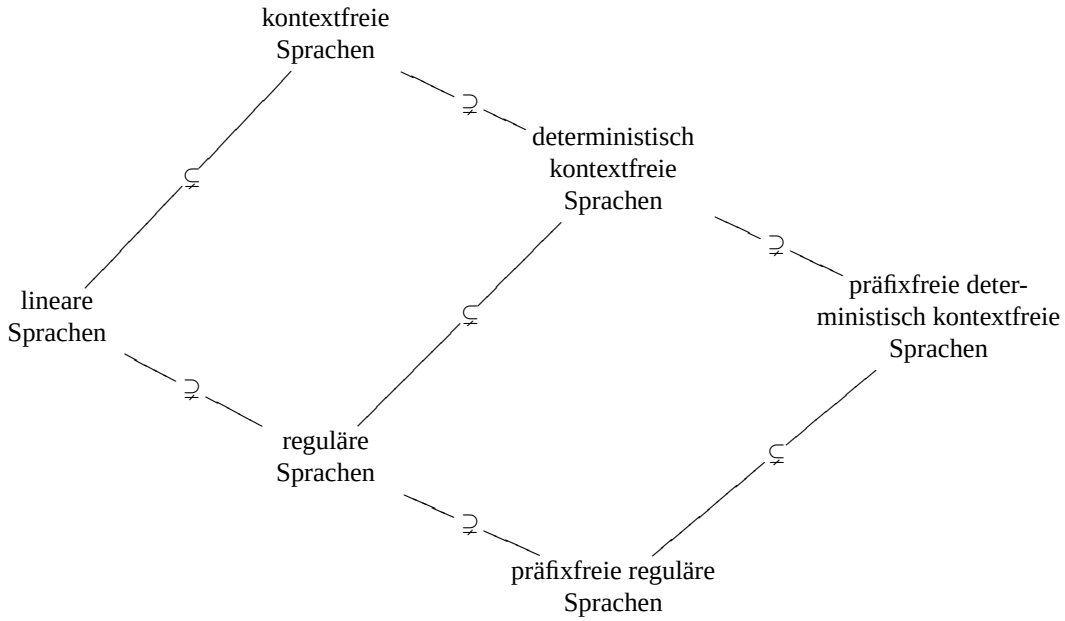


Abbildung 3.11. Überblick über die unterschiedlichen Sprachklassen.



Alle nicht explizit aufgeführten Inklusionen gelten nicht.

Abbildung 3.12. Überblick über die unterschiedlichen Sprachklassen.

Kapitel 4

Die Chomsky-Hierarchie

1 Überblick

In den beiden vorangegangenen Kapiteln haben wir zwei Sprachklassen (nebst einigen Spezialfällen) kennengelernt. Diese beiden Klassen ordnen sich in eine Hierarchie von insgesamt vier Sprachklassen ein, die erstmals von N. Chomsky vorgeschlagen wurde: die *Chomsky-Hierarchie*. Die Sprachklassen werden jeweils über ein Grammatik mit bestimmten syntaktischen Einschränkungen definiert und zu jeder Klasse gibt es ein naheliegendes Automatenmodell.
Zusätzlich zu den Klassen, die wir schon kennen, gibt es die *kontextsensitiven Sprachen* und die *aufzählbaren* Sprachen. Die kontextsensitiven Sprachen werden wir kurz in Abschnitt 2 ansprechen. Die aufzählbaren Sprachen werden in Kapitel 5 noch ausführlicher behandelt.
Die Chomsky-Hierarchie ist in Abb. 4.1 dargestellt.

> *In der Vorlesung wurde dieses Kapitel zurückgestellt und erst am Ende der Vorlesung besprochen. Deshalb wird hier auf manche Begriffe vorgegriffen (z.B. auf den Begriff der Turing-Maschine). Dies sollte aber mit den Kenntnissen aus den Grundstudiumsvorlesungen kein Problem sein.*

2 Kontextsensitive Sprachen

Die Idee hinter den *kontextsensitiven* bzw. *monotonen* Sprachen ist es, in den Grammatiken Produktionen der Form $\alpha A \beta \to \alpha \beta \gamma$ mit $|\gamma| \geq 1$ zuzulassen. Intuitiv erwartet man, daß jede kontextfreie Sprache auch kontextsensitiv ist – auch deshalb, weil wir eine Hierarchie von Sprachklassen bilden wollen. Allerdings gibt es dabei ein technisches Problem:

- Eine Produktion $A \to \varepsilon$ ist gemäß der obigen Bedingung nicht kontextsensitiv.

- Deshalb kann keine kontextsensitive Sprache kann das leere Wort enthalten, wenn wir bei der obigen Definition bleiben.

Daher gelten für ε einige Sonderregel.

Typ	**Name**	**Grammatiken**	**Automaten- bzw. Maschinenmodell**
Typ 0 Details: Kapitel 5 Seite 134	rekursiv aufzählbare Sprachen	- beliebig - $A_1 \dots A_n \rightarrow \alpha$ mit $n \geq 1$ - weitere Varianten	(D)TM = NTM
Typ 1 Details: Kapitel 4 Seite 103	kontext sensitive Sprachen	- kontextsensitiv[1] $\alpha A \beta \rightarrow \alpha \gamma \beta$ mit $\lvert\gamma\rvert \geq 1$ - monoton[1] (monoton) $\alpha \rightarrow \beta$ mit $\lvert\beta\rvert \geq \lvert\alpha\rvert$	TM mit linear beschränktem Band, LBA
Typ 2 Details: Kapitel 3 Seite 99	kontextfreie Sprechen	- kontextfrei $A \rightarrow \alpha$	(N)KA $\neq$ DKA
Typ 3 Details: Kapitel 2 Seite 50	regulären Sprachen	- regulär $A \rightarrow wB$ (rechtslinear) oder $A \rightarrow Bw$ (linkslinear)	(N)EA = DEA

Abbildung 4.1. Die Chomsky-Hierarchie

Definition 4.1 (Kontextsensitiv, monoton)

1. *Eine Grammatik G heißt* kontextsensitiv, *wenn jede Regel die Form $\alpha A\beta \to \alpha\gamma\beta$ mit $|\gamma| \geq 1$ hat.*

   Wenn das Axiom S auf keiner rechten Seite vorkommt, darf die Grammatik auch die Regel $S \to \varepsilon$ enthalten.

2. *Eine Grammatik G heißt* monoton *(nicht-verkürzend, beschränkt), wenn für jede Regel $\alpha \to \beta$ gilt $|\beta| \geq |\alpha|$.*

   Wenn das Axiom S auf keiner rechten Seite vorkommt, darf die Grammatik auch die Regel $S \to \varepsilon$ enthalten.

3. *Eine Sprache heißt* kontextsensitiv, *wenn sie von einer kontextsensitiven Grammatik erzeugt wird*

Achtung: Manchmal werden auch die Grammatiken, die wir monoton nennen, als kontextsensitive Grammatiken eingeführt! Der Grund dafür ist der nachfolgende Satz:

Satz 4.2 (Monotone Grammatiken und kontextsensitive Sprachen) *Eine Sprache ist genau dann kontextsensitiv, wenn es eine monotone Grammatik gibt, die sie erzeugt.*

Satz 4.3 (Entscheidbarkeit des Wortproblems) *Für ein Wort $w \in \Sigma^*$ und eine kontextsensitive Grammatik G ist entscheidbar, ob gilt $w \in L(G)$.*

Beweis: Idee: Systematische Generierung aller Wörter v aus $L(G)$ mit $|v| \leq |w|$. Dies ist möglich, da die Regeln monoton sind. □
Diese Idee motiviert auch das zu den kontextsensitiven Sprachen gehörige Automatenmodell: Turing-Maschinen, die nie über das linke bzw. rechte Ende der ursprünglichen Ausgabe hinauslaufen: *linear beschränkte Automaten* (LBA). Technisch begrenzt man die Eingabe w durch zwei Sonderzeichen: $¢w\$$. Wenn der Kopf auf $\$$ steht, macht er keine Rechtsbewegung, wenn er er auf $¢$ steht, macht er keine Linksbewegung.

Satz 4.4 (Linear beschränkter Automaten und kontextsenistive Sprachen) *Eine Sprache ist genau dann kontextsensitiv, wenn sie von einem linear beschränkten Automaten akzeptiert wird.*

Beispiel 4.1
Die Turing-Maschine, die wir in der Übung 11, Aufgabe 1b konstruiert haben, war (im Prinzip) ein LBA für die Sprache $a^n b^n c^n$.

Teil C

Entscheidbarkeit und Berechenbarkeit

Kapitel 5

Entscheidbare und aufzählbare Sprachen

1 Turing-Maschinen

Das Konzept der *Turing-Maschine* wurden von A. Turing eingeführt, um den Begriff des Berechenbaren bzw. des Entscheidbaren zu formalisieren und die Grenzen des Berechenbaren zu untersuchen. Etwa zur gleichen Zeit wurden andere Formalismen zur Charakterisierung des Berechenbaren vorgeschlagen, die sich alle als gleich mächtig erwiesen haben. Dies führte dann zur *Church'schen These*:

> Das was wir intuitiv als berechenbar auffassen, stimmt genau mit dem überein, was man mit Turing-Maschinen (oder mit den anderen äquivalenten Formalismen) berechnen kann (*Turing-Berechenbarkeit*).

Diese These ist prinzipiell nicht beweisbar! Aber (fast) alle Mathematiker und Informatiker glauben daran. In diesem Kapitel konzentrieren wir uns auf das Entscheidbare (und um das Aufzählbare). Um den Begriff des Berechenbaren kümmern wir uns erst im Kapitel 6. Das Konzept der Turing-Maschine ist aber schon im Hinblick auf den Begriff des (mechanisch) Berechenbaren gebildet:

- beliebig viel Platz (bzw. Papier) für Nebenrechnungen
- lokale Symbolmanipulationen
- Steuerung der Symbolmanipulationen durch eine (*endliche*) Berechnungsvorschrift (endliche Kontrolle)

Turing's Formalisierung:

- Die Maschine besitzt ein unendliches Band mit unendlich vielen Feldern zum Speichern von Symbolen.
- In jedem Berechnungsschritt kann die Maschine nur den Inhalt jeweils eines Feldes lesen und verändern. Dazu besitzt die Maschine einen Schreibe-/Lesekopf, der auf genau einem Feld steht.

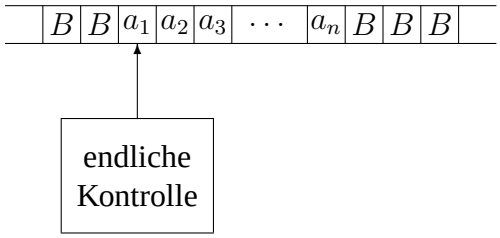


Abbildung 5.1. Schematische Darstellung einer Turing-Maschine.

- In einem Berechnungsschritt kann die Maschine den Kopf um ein Feld nach links oder rechts bewegen.
- Neben dem Band besitzt die Maschine eine eine endliche Menge von Zuständen. Der Zustand kann sich in jedem Berechnungsschritt ändern.
- Das in einem Berechnungsschritt auf das Band geschriebene Symbol, die Bewegungsrichtung des Kopfes und der Zustand nach dem Schritt hängt nur vom aktuellen Zustand und dem gelesenen Symbol ab.

Formal definieren wir eine Turing-Maschine wie folgt:

Definition 5.1 (Turing-Maschine, TM)
Eine Turing-Maschine M *über einem Alphabet* Σ *besteht aus:*

- *einer endlichen Menge von* Zuständen Q
- *einem* Bandalphabet Γ *mit* $\Gamma \supseteq \Sigma$, $Q \cap \Gamma = \varnothing$ *und* $B \in \Gamma \setminus \Sigma$, *wobei* B Blank *genannt wird,*
- *einer* Übergangsrelation $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{l, r\})$,
- *einem* Anfangszustand $q_0 \in Q$, *und*
- *einer Menge von* Endzuständen $F \subseteq Q$.

Wir notieren $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$.

Um die von einer Turing-Maschine akzeptierte Sprache zu definieren, bilden wir – wie bei Kellerautomaten – den Begriff der *Konfiguration* und definieren Übergänge zwischen den Konfigurationen. Dabei beschreibt eine Konfiguration den aktuellen Inhalt des Bandes, die Kopfposition auf dem Band und den aktuellen Zustand. Da wir in der Definition der Turing-Maschine $\Gamma \cap Q = \varnothing$ gefordert haben, können wir – wie bei Kellerautomaten – die Kopfposition durch die Position des Zustandes q in einer Zeichenreihe aus Γ^* repräsentieren.

Definition 5.2 (Konfiguration, Nachfolgekonfiguration)
Sei M *eine Turing-Maschine wie in Definition 5.1. Ein Wort* $\alpha q \beta$ *mit* $\alpha, \beta \in \Gamma^*$ *und* $q \in Q$ *heißt* Konfiguration *von* M.

1. *Für* $\beta = b_1 \dots b_n$ *mit* $n \geq 1$ *und* $((q, b_1), (p, b_1', r)) \in \delta$ *heißt* $\alpha b_1' p b_2 \dots b_n$ Nachfolgekonfiguration *von* $\alpha q \beta$.

2. *Für* $\alpha = a_1 \dots a_k$ *und* $\beta = b_1 \dots b_n$ *mit* $n, k \geq 1$ *und* $((q, b_1), (p, b'_1, l)) \in \delta$ *heißt* $a_1 \dots a_{k-1} p a_k b'_1 b_2 \dots b_n$ Nachfolgekonfiguration *von* $\alpha q \beta$.

3. *Für* $\beta = b_1 \dots b_n$ *mit* $n \geq 1$ *und* $((q, b_1), (p, b'_1, l)) \in \delta$ *heißt* $pBb'_1 b_2 \dots b_n$ Nachfolgekonfiguration *von* $q\beta$.

4. γ *heißt* Nachfolgekonfiguration *von* αq*, wenn* γ *Nachfolgekonfiguration von* $\alpha q B$ *ist.*

Wir schreiben $\gamma \vdash_M \gamma'$ *wenn* γ' *Nachfolgekonfiguration von* γ *(in* M*) ist. Wenn kein* γ' *mit* $\gamma \vdash_M \gamma'$ *existiert, schreiben wir* $\gamma \nvdash_M$

Obwohl das Band der Turing-Maschine unendlich ist, können wir jede Konfiguration der Turing-Maschine durch eine endliche Zeichenreihe darstellen. Denn wir gehen davon aus, daß am Anfang eine (endliche) Zeichenreihe aus Σ^* *auf dem Band steht; rechts und links davon stehen nur unendlich viele Blanks. Da in jedem Berechnungsschritt nur ein Feld des Bandes beschrieben werden kann, können also zu jedem Zeitpunkt nur endlich viele Zeichen auf dem Band stehen, die von* B *verschieden sind. Links und rechts von der Konfiguration können wir uns also jeweils eine unendliche Sequenz von Blanks denken, die wir aber nicht notieren. Bei der Definition der Übergangsrelation haben wir dies berücksichtigt: Ggf. werden die nötigen Blanks hinzugefügt (Fall 3 und 4 der Definition).*

Eine *Berechnung* einer Turing-Maschine für eine Eingabe $w \in \Sigma^*$ beginnt in der *Startkonfiguration* $q_0 w$, d. h. auf dem Band steht die Eingabe w, der Kopf steht auf dem ersten Zeichen der Eingabe, und die Turing-Maschine befindet sich im Anfangszustand q_0. Das Eingabewort wird von der Turing-Maschine akzeptiert, wenn es ausgehend von der Startkonfiguration eine Folge von Übergängen in eine Konfiguration gibt, in der sich die Turing-Maschine in einem akzeptierenden Zustand (d. h. einem Endzustand) befindet.
Etwas formaler definieren wir die akzeptierte Sprache einer Turing-Maschine wie folgt:

Definition 5.3 (Akzeptierte Sprache) *Sei* M *eine Turing-Maschine wie in Definition 5.1. Die von* M akzeptierte Sprache *ist definiert durch* $L(M) = \{w \in \Sigma^* \mid \alpha, \beta \in \Gamma^*, q \in F$ *mit* $q_0 w \vdash_M^* \alpha q \beta\}$.

Beispiel 5.1
Zur Verdeutlichung betrachten wir eine Turing-Maschine, die die Sprache $L(M) = \{a^n b^n | n \in \mathbb{N}\}$ akzeptiert. Der Turing-Maschine liegt die folgende Idee zugrunde: Wir inspzieren das Wort von außen nach innen und haken passende Paare von a's und b's von außen nach innen ab.
Die Berechung der Turing-Maschine bei Eingabe $aabb$ sieht dann wie folgt aus:

$$\begin{array}{llllllll} q_0aabb & \rightsquigarrow & \check{a}q_1abb & \rightsquigarrow & \check{a}aq_1bb & \rightsquigarrow & \check{a}abq_2b & \rightsquigarrow & \check{a}abbq_2 \\ & \rightsquigarrow & \check{a}abq_3b & \rightsquigarrow & \check{a}aq_4b\check{b} & \rightsquigarrow & \check{a}q_4ab\check{b} & \rightsquigarrow & q_4\check{a}ab\check{b} \\ & \rightsquigarrow & \check{a}q_0ab\check{b} & \rightsquigarrow & \check{a}\check{a}q_1b\check{b} & \rightsquigarrow & \check{a}\check{a}bq_2\check{b} & \rightsquigarrow & \check{a}\check{a}q_3b\check{b} \\ & \rightsquigarrow & \check{a}q_4\check{a}\check{b}\check{b} & \rightsquigarrow & \check{a}\check{a}q_0\check{b}\check{b} & \rightsquigarrow & \check{a}\check{a}\check{b}q_e\check{b} & & \end{array}$$

Obwohl die Idee relativ einfach ist, wird die formale Definition der Turing-Maschine schon recht aufwendig: $M = (Q, \Sigma, \Gamma, \delta, q_0, B, \{q_e\})$ mit $Q = \{q_0, q_1, q_2, q_3, q_4, q_e\}$, $\Gamma = \{a, b, \check{a}, \check{b}, B\}$

und

$\delta = \{((q_0, B), (q_e, B, r)),$	das leere Wort wird sofort akzeptiert
$((q_0, a), (q_1, \check{a}, r)),$	a abhaken und nun zugehöriges b suchen
$((q_1, a), (q_1, a, r)),$	die nicht abgehakten a's nach rechts überspringen
$((q_1, b), (q_2, b, r)),$	erstes nicht abgehaktes b's nach rechts überspringen
$((q_2, b), (q_2, b, r)),$	die weiteren nicht abgehakten b's nach rechts überspringen
$((q_2, B), (q_3, B, l)),$	wenn rechtes Ende gefunden, auf letztes b zurücksetzen
$((q_2, \check{b}), (q_3, \check{b}, l)),$	wenn abgehaktes b gefunden, auf letztes b zurücksetzen
$((q_3, b), (q_4, \check{b}, l)),$	letztes (nicht abgehaktes) b abhaken
$((q_4, b), (q_4, b, l)),$	nach links bis zum letzen abgehaken a laufen
$((q_4, a), (q_4, a, l)),$	nach links bis zum letzen abgehaken a laufen
$((q_4, \check{a}), (q_0, \check{a}, r)),$	jetzt steht der Kopf auf dem ersten nicht abgehakten a oder auf einem b oder einem $\check{b}$
$((q_0, \check{b}), (q_e, \check{b}, r))\}$	wenn der Kopf auf einem abgehakten b steht, wird die Eingabe akzeptiert

Die Turing-Maschine in obigem Beispiel ist deterministisch: Für jede Konfiguration gibt es höchstens eine Nachfolgekonfiguration. Im allgemeinen kann es aber mehrere Nachfolgekonfigurationen geben. Im folgenden werden wir überwiegend deterministische Turing-Maschinen betrachten, weil – wie wir sehen werden – die nicht-deterministischen Turing-Maschinen nicht mehr leisten können als die deterministischen Turing-Maschinen. Um das formulieren und beweisen zu können, mußtens wir jedoch erst einmal beide Varianten definieren.

Definition 5.4 (Deterministische Turing-Maschine, DTM) *Eine Turing-Maschine heißt* deterministisch, *wenn für ihre Übergangsrelation δ für alle $p, q, q' \in Q$, alle $X, Y \in \Gamma$ und alle $m, m' \in \{l, r\}$ mit*

$$((p, X), (q, Y, m)) \in \delta \qquad \textit{und} \qquad ((p, X), (q', Y', m')) \in \delta$$

gilt $q = q'$, $Y = Y'$ und $m = m'$.

Für jeden Zustand p und jedes gelesene Zeichen X gibt es also genau einen möglichen Übergang.

Achtung: Wenn wir nichts anderes sagen, verstehen wir unter einer Turing-Maschine immer eine deterministische Turing-Maschine (also TM $\hat{=}$ DTM). Wenn wir auch nicht-deterministische Turing-Maschinen betrachten, sagen wir dies explizit (NTM).

2 Entscheidbarkeit und Aufzählbarkeit

In der Vorlesung haben wir an verschiedenen Stellen die Frage gestellt, ob ein bestimmtes Problem entscheidbar ist. Beispiele dafür waren:

- Gilt $L(G_1) \cap L(G_2) = \varnothing$?
- Gilt $w \in L(G)$?
- Ist G mehrdeutig?
- Ist G eine LR(0)-Grammatik?

Dazu müssen wir zunächst den Begriff des *Problems* formalisieren. Formal ist ein Problem lediglich eine formale Sprache. Konzeptionell wird dies der Sache jedoch nicht ganz gerecht. Deshalb holen wir etwas weiter aus, um am Ende wieder bei der formalen Definition zu landen:

Probleminstanz Jedes Problem besitzt eine Menge von *Probleminstanzen*.

Eine konkrete Probleminstanz des Wortproblems für kontextfreie Sprachen sieht beispielsweise wie folgt aus: Geben ist die folgende kontextfreie Grammatik:

$$\begin{array}{rrcl} G: & S & \to & aA \,|\, aB \\ & A & \to & aA \,|\, a \\ & B & \to & aB \,|\, b \end{array}$$

und das Wort $w = aba$. Die Frage ist ob gilt $w \in L(G)$?

Repräsentation Jeder Probleminstanz läßt sich eine Repräsentation in einem festen Alphabet zuordnen.

Die obige Probleminstanz ließe sich beispielsweise wie folgt repräsentieren, wobei wir als Alphabet $\Sigma = \{V, \to, |, t, ;, :\}$ wählen:

$$\begin{array}{ll} V \to t|V|; & V \to t|V||; \\ V| \to t|V|; & V| \to t|; \\ V|| \to t|V||; & V|| \to t|| : \\ t|t||t| & \end{array}$$

Dabei werden die Variablen der Grammatik in Zeichenreihen $V| \ldots |$ und die Terminalzeichen in Zeichenreihen $t| \ldots |$ übersetzt. D. h.

$$\begin{array}{rcl} V & \hat{=} & S \\ V| & \hat{=} & A \\ V|| & \hat{=} & B \\ t| & \hat{=} & a \\ t|| & \hat{=} & b \end{array}$$

Die ersten drei Zeilen der obigen Repräsentation enstprechen also der Grammatik G, wobei die Regeln durch das Zeichen ';' getrennt sind. Die vierte Zeile repräsentiert das Wort w (getrennt durch das Zeichen ':').

Auf diese Weise läßt sich jede Probleminstanz des Wortproblems für kontextfreie Sprachen repräsentieren.

Wahre Probleminstanzen Jede Probleminstanz ist entweder *wahr* oder *falsch*.

Unser obiges Beispiel ist falsch, denn es gilt nicht $w \in L(G)$.

Problem Formal ist ein *Problem* die Sprache der Repräsentationen aller wahren Probleminstanzen des Problems.

Technisch gibt es also keinen Unterschied zwischen einem Problem und einer Sprache. Wir werden deshalb im folgenden den Begriff Sprache und Problem synonym verwenden. Den pragmatischen Unterschied sollten wir jedoch immer im Hinterkopf behalten.
Nachdem wir also den Begriff des Problems konzeptuell geklärt und technisch definiert haben, werden wir nun definieren, wann ein Problem (also eine Sprache) entscheidbar ist.

Definition 5.5 (Aufzählbarkeit und Entscheidbarkeit)
Ein Problem (eine Sprache) L heißt

1. aufzählbar, *wenn es eine deterministische Turing-Maschine M mit $L = L(M)$ gibt; sie heißt*

2. entscheidbar, *wenn es eine deterministische Turing-Maschine M mit $L = L(M)$ gibt, die auf jeder Eingabe hält (d.h für alle $w \in \Sigma^*$ gibt es ein γ mit $q_0 w \vdash_M^* \gamma \not\vdash_M$).*

Bemerkungen:

1. Eine Sprache ist also genau dann aufzählbar, wenn es eine (deterministische) Turing-Maschine, die diese Sprache akzeptiert. Die Turing-Maschine muß also auf den wahren Probleminstanzen irgendwann in einem akzeptierenden Zustand halten.

   Tatsächlich könnte die Turing-Maschine nach dem Erreichen eines akzeptierenden Zustandes sogar noch weiterarbeiten; sie muß in einem akzeptierenden Zustand nicht halten. Allerdings können wir jede Turing-Maschine so umbauen, daß sie beim Erreichen eines akzeptierenden Zustandes hält (ohne dabei die akzeptierte Sprache zu veränden). Deshalb sagen wir, daß eine Turing-Maschine in einem akzeptierenden Zustand hält, selbst wenn es noch einen möglichen Übergang gibt.

   Für falsche Probleminstanzen muß die Turing-Maschine aber nicht unbedingt halten (keine Antwort ist also auch eine Antwort und heißt bei Turing-Maschinen „nein"). Bei entscheidbaren Sprachen wird dagegen explizit gefordert, daß die Turing-Maschine auch für falsche Probleminstanzen immer hält (in einem nicht-akzeptierenden Zustand; dies interpretieren wir dann als „nein").

   Da also die Turing-Maschine für eine aufzählbare Sprachen nur im positiven Fall halten muß, werden aufzählbare Sprachen auch *semi-entscheidbar* genannt.

2. Es gibt eine äquivalente Charakterisierung der aufzählbaren Probleme: Ein *Generator* ist eine deterministische Turing-Maschine, die alle wahren Probleminstanzen auf ein besonderes Ausgabeband schreibt, getrennt durch ein ausgezeichnetes Zeichen # (vgl. Übungsblatt 11, Aufgabe 3). Dieser Generator *generiert* also alle Wörter der Sprache, bzw. *zählt sie auf*. Diese äquivalente Charakterisierung rechtfertigt die Bezeichnung *aufzählbar*.

3. **Achtung:** Abzählbar und aufzählbar haben verschiedene Bedeutung! Jedes Problem ist abzählbar (vergleiche Abschnitt 3.3 in Kapitel 1), da jede Sprache abzählbar ist. Aber es gibt Probleme, die nicht aufzählbar sind.

4. Entscheidbare Probleme werden manchmal auch *rekursiv* genannt; aufzählbare Probleme werden auch *rekursiv-aufzählbar* genannt.

Wir haben nun die aufzählbaren und die entscheidbaren Sprachen bzw. Probleme definiert. Noch wissen wir allerdings nicht, ob diese Klassen nicht doch zufälligeweise gleich sind. Per Definition ist jede entscheidbare Sprache aufzählbar; die Frage ist nun, ob es eine aufzählbare Sprache gibt, die nicht entscheidbar ist. Die Antwort ist: Ja, das *Halteproblem* ist aufzählbar, aber nicht entscheidbar. Damit ist die Unterscheidung der Begriffe aufzählbar und entscheidbar sinnvoll. Bevor wir aber die Aufzählbarkeit und die Unentscheidbarkeit des Halteproblems beweisen können, müssen noch das Handwerkszeug dafür bereitstellen.

3 Konstruktionen

Oft ist es mühselig, die Übergangsrelation einer Turing-Maschine explizit anzugeben[1]. Selbst einfache Algorithmen führen zu großen unüberschaubaren Übergangsrelationen (vgl. Beispiel 5.1). Deshalb geben wir hier einige Standardkonstruktionen an, aus denen wir dann komplexere Algorithmen aufbauen, ohne die Übergangsfunktion (explizit) anzugeben. In den meisten Fällen genügt es uns, hinreichend plausibel zu machen, daß eine Turing-Maschine für ein bestimmtes Problem existiert. Konstruieren werden wir diese Turing-Maschinen aber in den seltensten Fällen. Die folgenden Konstruktionen sollen ein Gefühl dafür geben, „was mit Turing-Maschinen“ geht.

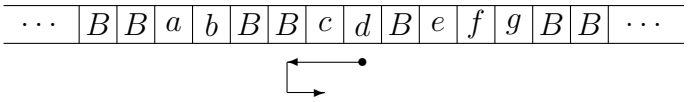
3.1 Suche des linken oder rechten Bandendes

Es gibt eigentlich kein Bandende; gemeint ist das Ende des bisher beschriebenen Bandes

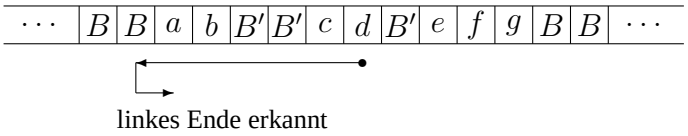
Idee: Bewege den Kopf nach links bzw. rechts bis zu einem Blank B.

Problem: Wenn ein Blank zwischen den bisher geschriebenen Zeichen vorkommt, dann wird dieses Blank fälschlicherweise als Bandende erkannt:

[1] Noch viel schwerer ist es, eine Turing-Maschine allein aus ihrer Übergangsrelation ohne erklärende Kommentare zu verstehen.



Lösung: Die Turing-Maschine schreibt niemals ein Blank B auf das Band. Stattdessen schreibt sie dann ein Sonderzeichen B'; auf diesem Sonderzeichen verhält sich die Turing-Maschine wie auf einem B. Zwischen den bisher geschriebenen Zeichen kann dann nur ein B' aber kein B vorkommen.



3.2 Einfügen von k neuen Zeichen $a_1 \ \ldots \ a_n$

Idee:

- Verschiebe den Bandinhalt links von der Kopfposition um k Zeichen nach links.

- Bewege den Kopf zur ursprünglichen Kopfposition zurück. Dazu muss man die die Position zuvor markieren (z. B. durch ein zusätzliches Sonderzeichen $\check{a}$ für jedes bisherige Bandsymbole a.

- Füge $a_1 \ \ldots \ a_n$ in die Lücke ein.

3.3 Sequenz, Alternative, Wiederholung

Turing-Maschinen können wir durch die üblichen Kontrollkonstrukte Sequenzen Alternative und Iteration zu neuen Turing-Maschinen kombinieren. Dazu muß man ihre Übergangsrelationen geeignet miteinander kombinieren.

> *Diese Kombinationen der Übergangsrelationen anzugeben, ist eine Fleißaufgabe, die an die Programmierung in Assembler erinnert und Ihnen vielleicht auch aus der Codegenerierung beim Compilebau bekannt ist. Wir verzichten hier aus Zeitgründen darauf, diese Kombinationen zu formulieren.*

Abbildung 5.2 stellt die entsprechenden Konstrukte schematisch dar. Weitere Konstrukte werden wir bei Bedarf ad hoc benutzen. In dieser Darstellung bedeutet „ja“, daß die Turing-Maschine einen akzeptierenden Zustand erreicht hat. „nein“ bedeutet, daß es keine Nachfolgekonfiguration gibt und der aktuelle Zustand kein akzeptierender Zustand ist.

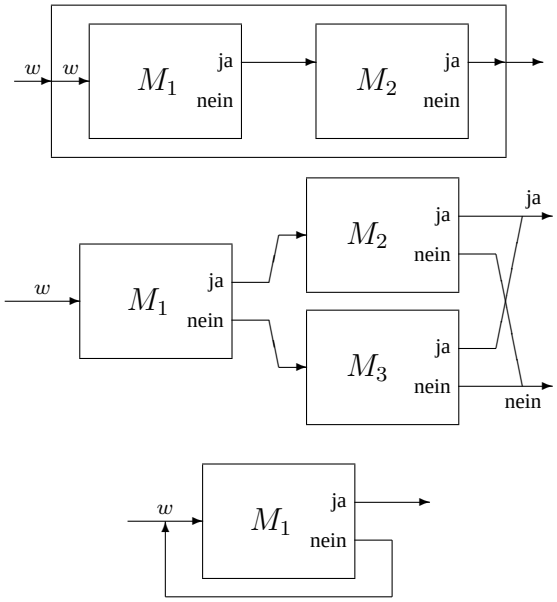


Abbildung 5.2. Schematische Darstellung: Sequenz, Alternative und Iteration.

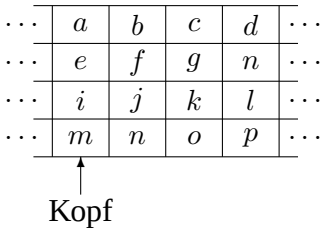


Abbildung 5.3. Eine Mehrspur-Turing-Maschine

3.4 Mehrspurmaschinen

Bei einer *Mehrspur-Turing-Maschine* ist das Band in mehrere Spuren unterteilt (vgl. Abb. 5.3). Alle Spuren werden gleichzeitig gelesen und geschrieben. Die Eingabe steht auf der ersten Spur. Mehrspur-Turing-Maschinen erlauben also eine gewisse Strukturierung der Daten auf dem Band.

Tatsächlich sind Mehrspur-Turing-Maschinen keine Verallgemeinerung des bisherigen Konzepts der Turing-Maschine. Wir müssen lediglich das Bandalphabet geeignet wählen: $\Gamma = \Gamma_1 \times \Gamma_2 \times \cdots \times \Gamma_n$, wobei $\Gamma_1, \Gamma_2 \ldots, \Gamma_n$ die Alphabete der einzelnen Spuren sind. Da die Eingabe auf der ersten Spur steht, ist die folgende Annahme zweckmäßig: für $a \in \Gamma_1$ gilt $a \mathrel{\hat{=}} (a, B, \ldots, B)$ und $B \mathrel{\hat{=}} (B, B, \ldots, B)$. Dadurch entspricht eine Eingabe $a_1 \ldots a_n$ dem Bandinhalt $(a_1, B, B, B) \ldots (a_n, B, B, B)$, d. h. die erste Spur enthält genau die Eingabe; alle anderen Spuren sind leer bzw. mit Blanks beschriftet.

Mehrband-Turing-Maschinen Einer *Mehrband-Turing-Maschine* besitzt mehrere Bänder. Im Gegensatz zu Mehrspur-Turing-Maschinen können sich jedoch die Köpfe auf jedem Band unabhängig voneinander bewegen (vgl. Abb. 5.4). Beispielsweise kann sich in einem Schritt der Kopf auf dem ersten Band nach links und der Kopf auf dem zweiten Band nach rechts bewegen.

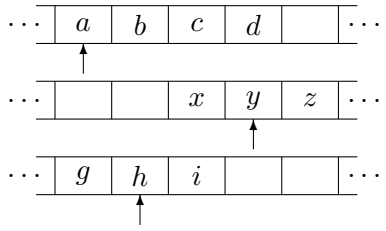


Abbildung 5.4. Eine Mehrband-Turing-Maschine

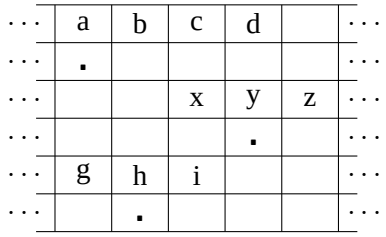


Abbildung 5.5. Repräsentation von n Bändern durch $2n$ Spuren

Die Eingabe steht am Anfang auf einem ausgezeichneten Eingabeband (meist das erste Band). Eine Mehrband-Turing-Maschine mit n Bändern (*n-Bandmaschine*) können wir durch eine Mehrspur-Turing-maschine mit $2n$ Spuren (*$2n$-Spurmaschine*) simulieren. Dazu wird werden die n Bänder durch $2n$ Spuren simuliert, wie dies in Abb. 5.5 dargestellt ist: In den geraden Spuren ist der Bandinhalt repräsentiert; in den ungeraden Spuren darunter ist durch das Sonderzeichen **.** die Kopfposition des jeweils darüberliegenden Bandes markiert.
Ein Schritt der Mehrband-Turing-Maschine können wir dann wie folgt simulieren:

- Durchlaufe das bisher beschriebene Band von links nach rechts und sammle dabei die Zeichen der n Bänder an den aktuellen Kopfpositionen ein und speichere sie im Zustand.
- Führe in einem Durchlauf von rechts nach links auf jedem Band die nötige Änderung an der jeweiligen Kopfposition durch (Schreiben des neuen Zeichens und der Markierung der neuen Kopfposition auf der separaten Spur).
- Gehe in den nächsten Zustand über und beginnen mit der Simulation des nächsten Schrittes.

Bei beiden Durchläufen muss im Extremfall das gesamte bisher beschriebene Band abgefahren werden. Das ist zwar sehr aufwendig – aber funktioniert.

> *Wenn man sich geschickt anstellt, muss man das Band jedoch nicht zweimal durchlaufen, um einen Schritt der Mehrband-Turing-Maschine zu simulieren.*

3.5 Parallele Ausführung zweier Turing-Maschinen

Um eine Eingabe w von zwei Turing-Maschinen M_1 und M_2 gleichzeitig abarbeiten zu lassen, kopiert man die Eingabe auf zwei verschiedene Bänder. Dann startet man die Turing-Maschine M_1 auf dem einen und die Turing-Maschine M_2 auf anderen Band. Wenn M_1 oder M_2 “ja” sagen (also die Eingabe w akzeptieren), dann ist auch das Gesamtergebnis “ja”. Nur wenn sowohl M_1 als auch M_2 “nein” sagen, ist auch das Gesamtergebnis “nein”.

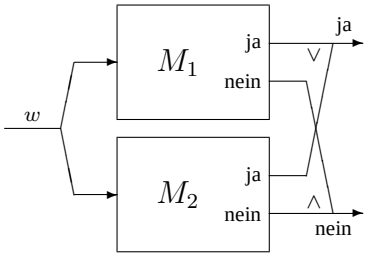


Abbildung 5.6. Parallele Ausführung zweier Turing-Maschinen.

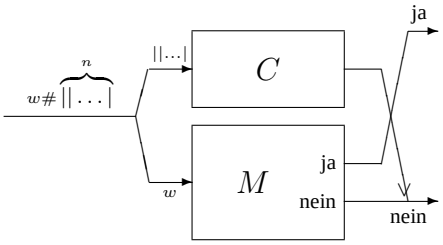


Abbildung 5.7. Beschränkte Simulation $Sim(M)$ eine Turing-Maschine M.

3.6 Beschränkte Simulation

Manchmal möchte man wissen, ob eine bestimmte Turing-Maschine M ein Eingabewort w innerhalb von n Berechnungsschrittem akzeptiert. Dazu erweitern wir die Turing-Maschine um einen Zähler (Striche auf einem separaten Band), der bei jedem Berechnungsschritt um eins verringert wird (ein Strich wird gelöscht). Wenn das Eingabewort w akzeptiert wird bevor der Zähler auf Null steht, akzeptiert die modifizierte Turing-Maschine, ansonsten akzeptiert sie nicht. Diese Turing-Maschine nennen wir *beschränkten Simulator* $Sim(M)$ von M. Als Eingabe erhält sie das Wort w und die Zahl n getrennt durch ein Sonderzeichen $\#$.
In der Abbildung 5.7 ist dieser beschränkte Simulator schematisch dargestellt. Die Turing-Maschine C (C steht für engl. counter), die für diese Konstruktion benötigt wird, streicht in jedem Schritt einen Strich; wenn kein Strich mehr vorhanden ist, hält die Maschine an. Dabei laufen C und M synchron; C macht immer dann einen Schritt wenn M einen Schritt durchführt. Wenn kein Strich mehr vorhanden ist, bleiben beide Maschinen stehen

3.7 Nicht-deterministische Turing-Maschinen

Die Idee der beschränkten Simulation können wir etwas modifizieren, um zu beweisen, dass wir jede nicht-deterministische Turing-Maschin durch eine deterministische Turing-Maschine simulieren können. Nicht-deterministische Turing-Maschinen leisten also nicht mehr als deterministische Turing-Maschinen.

Mit etwas guten Willen können wir also auch den Nicht-determinismus als eine Konstruktion auffassen. Wem das nicht so gut gefällt, kann diesen Abschnitt als Einschub auffassen, der wegen der benutzten Beweistechnik gut an diese Stelle paßt.

Beweisskizze: Wir skizzieren hier nur die wesentliche Idee: Dazu betrachten wir eine beliebige nicht-deterministische Turing-Maschine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. Für diese Turing-Maschine konsturieren wir nun eine deterministische Turing-Maschine, die genau die gleiche Sprache akzeptiert.
Für einen Zustand $q \in Q$ und ein gelesenens Zeichen $a \in \Gamma$ definieren wir $A_{(q,a)} = \{(p, b, m) \mid ((q, a), (p, b, m)) \in \delta\}$ als die Menge der Alternativen (d. h. der möglichen Übergänge) in Zustand q bei gelesenem Eingabezeichen a. Die maximale Anzahl k von Alternativen der Turing-Maschine M ergibt sich dann durch $k = max\{|A_{(q,a)}| \mid q \in Q, a \in \Gamma\}$ (da Q, Γ und δ endlich sind, gilt $k \in \mathbb{N}$).
Wir können nun jeder Alternative aus $A_{(q,a)}$ eindeutig eine Nummer aus $\{1, 2, \ldots, k\}$ zuordnen. Sei nun $Sim(M)$ eine Turing-Maschine, die eine Eingabe der Form $w\#\pi$ erhält, wobei $w \in \Sigma^*$, $\# \notin \Sigma$ und $\pi = k_1 k_2 \ldots k_n$ mit $k_i \in \{1, 2, \ldots, k\}$ ist. $Sim(M)$ simuliert die Turing-Maschine M auf der Eingabe w für $|\pi|$ Schritte; k_i gibt an, welche Alternative der Turing-Maschine M im i-ten Simulationsschritt von Sim(M) ausgewählt werden soll. Wenn diese Alternative im Schritt i-ten nicht möglich ist, wird die Simulation (erfolglos) abgebrochen.

Beobachtungen: $Sim(M)$ terminiert für jede Eingabe $w\#\pi$ nach $|\pi|$ Schritten. Außerdem ist $Sim(M)$ deterministisch, denn π schreibt für $Sim(M)$ die aus M zu wählende Alternative genau vor.
Nun können wir aus $Sim(M)$ eine Turing-Maschine M' bauen, die für eine Eingabe w systematisch alle Paare $w\#\pi$ erzeugt und dann $Sim(M)$ mit Eingabe $w\#\pi$ startet. Wenn $Sim(M)$ für ein π die Eingabe $w\#\pi$ akzeptiert, dann akzeptiert M' die Eingabe w. Offensichtlich gilt $L(M) = L(M')$. Denn es gilt $w \in L(M')$ gdw. ein $\pi \in \{1, 2, \ldots, k\}^*$ mit $w\#\pi$ von $Sim(M)$ akzeptiert wird gdw. $w \in L(M)$ gilt.
Mit dieser Konstruktion haben wir also bewiesen, daß nicht-deterministische Turing-Maschinen keinen Zugewinn an Audruckskraft bringen; und zwar weder was die aufzählbaren noch was die entscheidbaren Sprachen betrifft (letzteres müssen wir aber noch beweisen, Punkt 2).

Satz 5.6 (NTM $\equiv$ DTM)

1. *Für jede nicht-deterministische Turing-Maschine M gibt es eine deterministische Turing-Maschine M' mit $L(M) = L(M')$.*

2. *Darüber hinaus können wir M' so konstruieren, daß M' auf allen Eingaben hält, wenn M keine unendliche Berechnung besitzt.*

Beweis:

1. Formalisieren der obigen Idee.

2. M' muß bei der systematischen Generierung aller π nur solche π konstruieren, für die $Sim(M)$ die Simulation nicht vorzeitig abgebrochen hat (weil die zu wählende Alternative im Schritt i nicht möglich war).

   Fortsetzungen von vorzeitig abgebrochenen π müssen nicht generiert werden, da auch sie vorzeitig abgebrochen würden. Wenn sich M' alle diese π merkt (z. B. auf einem

separaten Band), und M keine unendliche Berechnung besitzt, muss M irgendwann kein neues π generieren und kann halten (ohne zu akzeptieren).

□

Am oberen Ende und am unteren Ende der Chomsky-Hierarchie sind also die nicht-deterministischen Automatenmodelle nicht mächtiger als die entsprechenden deterministischen (DEA ≡ NEA und DTM ≡ NTM).
Für die Kellerautomaten, die dazwischen liegen, gilt dies jedoch nicht. Mit nicht-deterministische Kellerautomaten kann man echt mehr Sprachen akzeptieren als mit deterministischen Kellerautomaten. Das sollte uns zumindest beim ersten mal verblüffen.
Noch erstaunlicher ist, daß wir für die linear beschränkten Automaten (LBA) bis heute nicht wissen, ob die deterministische Variante genausoviel kann wie die nicht-deterministische.

4 Erweiterungen und Einschränkungen

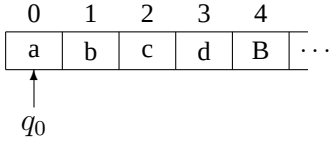
Es gibt viele verschiedene Varianten von Turing-Maschinen. Manche davon verallgemeinern die von uns vorgestellte Variante, andere schränken sie ein, wieder andere sind zunächst unvergelichbar. Allerdings stellt sich heraus, daß keine Verallgemeinerung mehr berechnen kann als die von uns definierte Variante. Dies ist ein deutliches Indiz für die eingangs erwähnte Church'sche These.

1. *Natürlich lassen sich leicht Verallgemeinerungen von Turing-Maschinen definieren, die mehr können als Turing-Maschinen. Allerdings besitzen diese Verallgemeinerungen dann einen Mechanismus, den wir intuitiv nicht als berechnenbar auffassen. Ein Beispiel dafür sind sog. Orakel-Turing-Maschinen.*
2. *Obwohl die hier vorgestellten Varianten bezüglich ihrer Berechenbarkeit gleich mächtig sind, gibt es Unterschiede bezüglich der Komplexität von Berechnungen. Natürlich lassen sich manche Berechnungen auf einer parallelen Turing-Maschine schneller durchführen als auf einer sequentiellen Turning-Maschine. Diese Unterschiede werden in der Komplexitätstheorie genauer untersucht.*

Die folgenden Beispiele sollen ein Gefühl dafür vermitteln, welche Verallgemeinerungen und Einschränkungen von Turing-Maschinen existieren. Außerdem zeigen diese Beispiel mit welchen Techniken man die Äquivalenz verschiedener Varianten beweisen kann.

4.1 Einseitig beschränktes Band

Bei unsere Turing-Maschine war das Band nach links und rechts unbeschränkt; es hat also keinen Anfang und kein Ende. Nun betrachten wir eine Turing-Maschine mit einem Band, das einen Anfang hat, aber kein Ende; es ist also einseitig beschränkt. Dies ist nachfolgend graphisch dargestellt (das Band ist nach rechts unbeschränkt):



Die Turing-Maschine kann ihren Kopf also nicht nach links über das erste Feld hinaus bewegen. Alle anderen Übergänge sind wie bei der klassischen Turing-Maschine möglich, wobei wir davon ausgehen, daß der Kopf initial auf dem ersten Feld des Bandes steht.
Trotz der Einschränkung kann man mit einer einseitig beschränkten Turing-Maschine dieselben Sprachen akzeptieren (und dieselben Funktionen berechnen) wie mit einer gewöhnlichen Turing-Maschine.
Um dies zu beweisen, müssen wir das Verhalten einer beidseitig unbeschränkten (gewöhnlichen) Turing-Maschine durch eine einseitig beschränkte Turing-Maschine simulieren. Dazu „falten“ wir den linken Teil des Bandes nach rechts, so daß (mit Ausnahme des ersten Feldes) auf jedem Feld zwei Symbole stehen: Oben stehen die Sybole der linken Bandhälfte; unten stehen die Symbole der rechten Bandhälfte. Dies Abbildung ist nachfolgend graphisch dargestellt:

	-3	-2	-1	0	1	2	3	
···	B	B	B	a	b	c	B	···

Γ

$\downarrow$

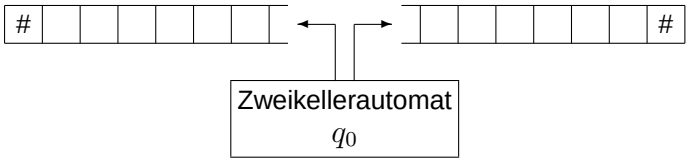
0	1	2	3	4	5	6	7	
a	B	B	B	B	B	B	B	···
	b	c	d	B	B	B	B	···

$$\Gamma' = (\Gamma \times \Gamma) \cup \Gamma$$

Es ist klar, daß wir nun alle Bewegungen der beidseitig unbeschränkten Turing-Maschine auf dem einseitig beschränktem Band simulieren können. Die Turing-Maschine muß sich lediglich im Zustand merken, ob sie gerade auf der linken (oberen) oder rechten (unteren) Bandhälfte arbeitet. Beim ersten Feld muß sie dies entsprechend der Bewegungsrichtung umschalten.
Eine mathematische Definition der Turing-Maschine mit einseitig unbeschränktem Band und der oben angedeuteten Simulation ist nicht kompliziert; sie erfordert aber etwas Sorgfalt.

4.2 Kellerautomaten mit zwei Kellern

Man kann sich leicht überlegen, daß man mit einer Turing-Maschine auch einen Kellerautomaten simulieren kann: Auf der linken Seite des Bandes kann man den Keller simulieren; auf der rechten Seite liest man die Eingabe. Mit Turing-Maschinen kann man natürlich noch mehr Sprachen akzeptieren als mit Keller-Automaten z.B. die Sprache „$a^n b^n c^n$“ (vgl. Übung 11, Aufgabe 1b).
Wenn wir allerdings einen *Kellerautomaten mit zwei Kellern* ausstatten, auf die der Kellerautomat unabhängig voneinander zugreifen kann, dann können wir mit einem solchen „Zweikellerautomaten“ eine Turing-Maschine simulieren. Denn die beiden Keller können wir zu einem beidseitig unendlichen Band zusammenkleben:



Ein Kellerautomat mit zwei Kellern kann also eine Turing-Maschine simulieren. Manche Operationen, beispielsweise das Einfügen von Zeichen, lassen sich hiermit sogar einfacher realisieren als mit einer gewöhnlichen Turing-Maschine.

> *Wir haben ja gesehen, daß Turing-Maschinen mit einseitig beschränktem Band genauso mächtig sind wie gewöhnliche Turing-Maschinen. Da drängt sich doch der Verdacht auf, daß man mit einem gewöhnlichen Kellerautomaten (mit einem Band) auch eine einseitig beschränkte Turing-Maschine simulieren können müßte. Damit wären Kellerautomaten und Turing-Maschinen äquivalent. Natürlich stimmt das nicht! Wo liegt der Denkfehler?*

4.3 Zweidimensionale Turing-Maschinen

Gewöhnliche Turing-Maschinen arbeiten mit einem Band. Wenn man über das Gebiet der Formalen Sprachen zu den Turing-Maschinen gelangt, erscheint dies naheliegend. Wenn man jedoch Turing-Maschinen als das abstrakte Modell dessen betrachte, was man auf einem Blatt Papier berechnen kann, dann ist dies nicht mehr so naheliegend. Denn das Blatt Papier auf dem man rechnet, ist zweidimensional.
Demenstsprechend betrachten wir nun Turing-Maschinen, die auf einem (in alle vier Richtungen) unbeschränkten karierten Blatt Papier arbeitet. Der Kopf der Turing-Maschine steht auf einem „Kästchen“ dieses Papieres und kann sich in je der vier Richtungen bewegen. Eine solche Turing-Maschine heißt *zweidimensional*.

> *Tatsächlich ist diese Erweiterung auch für drei, vier oder noch mehr Dimensionen möglich; man spricht dann von* mehrdimensionalen Turing-Maschinen.

Auch zweidimensionale Turing-Maschinen kann man durch eine gewöhnliche (eindimensionale) Turing-Maschine simulieren. Dazu kann man das zweidimensionale Papier auf ein eindimensionales Band abbilden, wie dies nachfolgend angedeutet ist:

	-3	-2	-1	0	1	2	3
3							
2							
1					d	e	
0				a	b	c	
-1			f	g	h		
-2					i		
-3							

$$\downarrow$$

$$\ldots \# \blacksquare Bde\# \blacksquare abc\# f \blacksquare gh\# \blacksquare Bi\# \ldots$$

Diese Abbildung ähnelt der Abbildung eines mehrdimensionalen Arrays auf den linearen Speicher. Man benötigt zwei Sonderzeichen. Ein Sonderzeichen (#) trennt die Zeichenketten der einzelnen Zeilen voneinander. Das andere Sonderzeichen (**.**) markiert die 0-Position innerhalb der Zeile, um die Ausrichtung der Zeilen zueinander zu ermöglichen.
Die Simulation der Bewegungen der zweidimensionalen Turing-Maschine ist relativ aufwendig, denn bei einer Bewegung nach oben oder unten muß man die richtige Position in der entsprechenden Zeile suchen. Dazu braucht man noch einen zusätzlichen Zähler (den kann man auf einem zusätzlichen Band realisieren).

5 Eigenschaften entscheidbarer und aufzählbarer Sprachen

Ähnlich wie bei den regulären und kontextfreien Sprachen unterscuhen wir nun die Eigenschaften der entscheidbaren und aufzählbaren Sprachen. Außerdem untersuchen wir den Zusammenhang zwischen aufzählbaren und entscheidbaren Sprachen.

Satz 5.7 (Abschlußeigenschaften)

1. *Die entscheidbaren Sprachen sind unter den folgenden Operationen abgeschlossen:*
   - *Komplementbildung,*
   - *Vereinigung und*
   - *Durchschnitt.*
2. *Die rekursiv aufzählbaren Sprachen sind unter den folgenden Operationen abgeschlossen:*
   - *Vereinigung und*
   - *Durchschnitt.*

Beweis: Wir geben hier nur die grobe Idee der Beweise für die Aussagen an:

1. Seien L_1 und L_2 entscheidbare Sprachen und M_1 und M_2 zwei Turing-Maschinen, die die Sprache L_1 bzw L_2 akzeptieren und auf jeder Eingabe halten.
   - Komplementbildung: Wir vertauschen in M_1 die „Ausgaben“ “ja” und “nein”; dies ist möglich, da per Annahme die Turing-Maschine M_1 auf jeder Eingabe hält. Offensichtlich akzeptiert die modifizierte Turing-Maschine die Sprache $\overline{L_1}$; außerdem hält sie auf jeder Eingabe.
   - Vereinigung: Wir schalten die Turingmaschinen M_1 und M_2 parallel (vgl. 3.5); diese Maschine akzeptiert dann die Sprache $L_1 \cup L_2$ und hält auf jeder Eingabe.
   - Durchschnitt: De Morgan mit Abschluß unter Komplement und Vereinigung. (Man kann aber auch relativ einfach eine direkte Konstruktion angeben.)
2. - Vereinigung: Wie oben.
   - Durchschnitt: Explizite Konstruktion (Parallelschaltung, die nur dann akzeptiert, wenn beide Maschinen akzeptieren).

□

Bemerkung: Die aufzählbaren Sprachen sind nicht abgeschlossen unter Komplementbildung, denn sonst wären sie entscheidbar (vergleiche Satz 5.8). Wir werden später sehen, dass es aber aufzählbare Sprachen gibt, die nicht entscheibar sind. Beispielsweise ist das Wortproblem gemäß Satz 5.13 aufzählbar, aber gemäß Satz 5.9 nicht entscheidbar.
Die Entscheidbarkeit einer Sprache können wir über die Aufzählbarkeit der Sprache und ihres Komplementes charakterisieren. Denn, wenn wir eine Turing-Maschine haben, die im positive Falle „ja“ sagt (und terminiert) und eine, die im negativen Falle „ja“ sagt (und terminiert), können wir daraus eine Turing-Maschine bauen, die im positiven Falle „ja“ sagt und im negativen Falle „nein“ sagt, und in beiden Fällen terminiert.

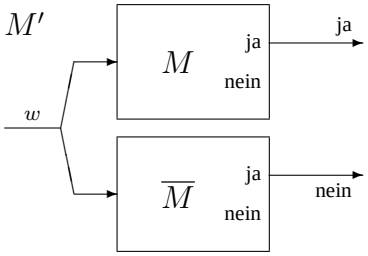
Satz 5.8 (Entscheidbarkeit und Aufzählbarkeit)
Eine Sprache L ist genau dann entscheidbar, wenn sowohl L auch als $\overline{L}$ aufzählbar sind.

Beweis:

„⇒“ Klar (für die Aufzählbarkeit von $\overline{L}$ können wir Satz 5.7 1.a anwenden).

„⇐“ Seien nun L un $\overline{L}$ aufzählbar. Dann gibt es eine Turing-Maschine M, die L akzeptiert und eine Turing-Maschine $\overline{M}$, die $\overline{L}$ akzeptiert (diese Maschinen müssen nicht unbedingt auf allen Eingaben halten).

Aus diesen beiden Turing-Maschinen konstruieren wir nun eine Turing-Maschine M', die auf allen Eingaben hält:



In M' sind beide Turing-Maschinen M und $\overline{M}$ parallel geschaltet; wenn eine der beiden Maschinen akzeptiert, hält M' an. Allerdings akzeptiert M' nur die Wörter aus L, Wörter, die von $\overline{M}$ akzeptiert werden, werden verworfen. Offensichtlich gilt dann $L(M') = L(M) = L$ und M' hält auf jeder Eingabe.

□

Eine unmittelbare Konsequenz dieses Satz ist die folgende Aussage: Wenn eine Sprache aufzählbar aber nicht entscheidbar ist, dann ist ihr Komplement nicht aufzählbar.

6 Unentscheidbare Probleme

Bereits in der Einleitung haben wir uns überlegt, daß es unentscheidbare Sprachen geben muß. Denn es gibt überabzählbar viele Sprachen, aber nur abzählbar viele Turing-Maschinen. Dieses

Argument ist jedoch unbefriedigend, da es kein konkretes Beispiel für eine nicht entscheidbare Sprache liefert. In diesem Abschnitt werden wir zunächst ein konkretes Beispiel für ein unentscheidbares Problem kennenlernen: das Wortproblem für Turing-Maschinen.
Wie in vielen Unmöglichkeitsbeweisen, werden wir die Unentscheidbarkeit des Wortproblems mit Hilfe eines Diagonalisierungsarguments beweisen. Wenn wir erst einmal ein unentscheidbares Problem haben, können wir relativ einfach die Unentscheidbarkeit weiterer Probleme beweisen; durch *Reduktion*. Wir werden sogar noch ein viel weitreichenderes Resultat kennenlernen, daß im wesentlichen besagt: Der erste Satz von Rice besagt, daß man über die von einer Turing-Maschine akzeptierte Sprache praktisch nichts entscheiden kann.

6.1 Das Wortproblem

Das *Wortproblem für Turing-Maschinen* besteht aus der Frage, ob eine bestimmte Turing-Maschine M ein bestimmtes Wort w akzeptiert oder nicht; also ob $w \in L(M)$ gilt. Um das Wortproblem zu präzisieren, müssen wir die Probleminstanzen, also die Paarr (M, w) syntaktisch repräsentieren. Dazu müssen wir uns eine Repräsentation für Turing-Maschinen und die Eingabewörter überlegen. Wir beginnen mit der Repräsentation der Turing-Maschinen.

Repräsentation einer Turing-Maschine Jede Turing-Maschine läßt sich durch eine Zeichenreihe über dem Alphabet $\Sigma = \{q, t, |, \rightarrow, l, r, \#\}$ repräsentieren. Dazu numerieren wir zunächst die Zustände durch; die Zustandsnummer repräsentieren wir dann durch eine Anzahl von Strichen. Entsprechend numerieren wir das Bandalphabet durch. Dabei gehen wir davon aus, daß q_0 der Anfangszustand ist und B das erste Zeichen des Bandalphabetes ist.
Demenstprechend ergibt sich die folgende Repräsentation für Zustände und Zeichen des Bandalphabetes:

$$\begin{array}{llll} q_0 & \rightsquigarrow & q & \qquad B \rightsquigarrow t \\ q_1 & \rightsquigarrow & q| & \qquad a \rightsquigarrow t| \\ q_2 & \rightsquigarrow & q|| & \qquad b \rightsquigarrow t|| \\ q_3 & \rightsquigarrow & q||| & \qquad c \rightsquigarrow t||| \\ \vdots & & & \qquad \vdots \end{array}$$

Um die Übergangsrelation δ der Turing-Maschine zu repräsentieren, können wir der Reihe nach jeden einzelnen Übergang von δ aufzählen. Ein einzelner Übergang können wir wie folgt übersetzen:

$((q_3, b), (q_2, c, l)) \in \delta \quad \rightsquigarrow \quad q|||t|| \rightarrow q||t|||l$

Da am Ende jedes Übergangs immer entweder ein l oder ein r steht, müssen wir kein spezielles Sonderzeichen zur Trennung der verschiedenen Übergänge einfügen; wir schreiben sie einfach unmittelbar hintereinander.
Die Menge der Endzustände der Turing-Maschine können wir wieder durch explizites Aufzählen angeben. Dies könnte wie folgt aussehen:

$F = \{q_1, q_3\} \quad \rightsquigarrow \quad q|q|||$

Insgesamt wird eine Turing-Maschine also durch eine Aufzählung aller ihrer Zustände, gefolgt von der Aufzählung ihres Bandalphabetes, gefolgt von der Aufzählung aller Übergänge, gefolgt

von der Aufzählung der Endzustände repräsentiert – jeweils getrennt durch das Sonderzeichen $\#$.
Für eine Turing-Maschine M bezeichnet dann $\langle M \rangle$ diese Repräsentation. Die folgenden Ergebnisse sind jedoch unabhängig von der hier gewählten Repräsentation.

> *Wichtig ist im folgenden nur, daß es eine festgelgte syntaktische Repräsentation für jede Turing-Maschine gibt.*

Repräsentation der Eingabe Entsprechend kann jedes Eingabewort w repräsentiert werden. Wir bezeichnen diese Repräsentation mit $\langle w \rangle$. Ein Beispiel hierfür ist:

$$abcaa \;\rightsquigarrow\; t|t||t|||t|t|$$

Formalisierung des Problems Mit diesen Repräsentationen können wir nun das Wortproblem formalisieren. Da wir später auch das Halteproblem betrachten werden formalisieren wir es an dieser Stelle gleich mit.
Das *Wortproblem*:

$$L_W = \{\langle M \rangle \# \langle w \rangle \mid M \text{ ist eine (det.) Turing-Maschine und } w \in L(M)\}$$

Das *Halteproblem*:

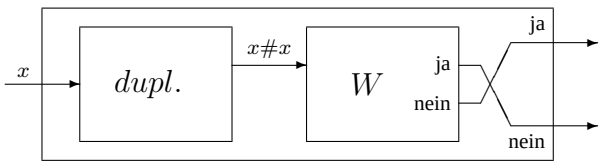
$$L_H = \{\langle M \rangle \# \langle w \rangle \mid M \text{ ist eine (det.) Turing-Maschine und hält bei der Eingabe } w\}$$

6.2 Unentscheidbarkeit des Wortproblems

Da wir das Wortproblem nun formalisiert haben, können uns nun Gedanken darüber machen, ob es entscheidbar ist. Es ist nicht entscheidbar:

Satz 5.9 (Unentscheidbarkeit des Wortproblems für Turingmaschinen)
Das Wortproblem L_W ist nicht entscheidbar.

Beweis: Wir beweisen die Aussage durch Widerspruch. Wir nehmen also an, daß das Wortproblem entscheidbar ist. D. h. wir nehmen an, daß es eine deterministische Turing-Maschine W mit $L(W) = L_W$ gibt, die für alle Eingaben hält.
Wir werden im folgenden aus der Existenz von W einen Widerspruch konstruieren. Dazu konstruieren wir aus W die deterministische Turingmaschine M wie in der nachfolgenden Abbildung dargestellt:



Diese Turing-Maschine M produziert aus ihrer Eingabe x zunächst das Wort $x\#x$ (wir nennen diesen Teil Duplizierer); dann startet M auf dem Wort $w\#x$ die Turingmaschine W; am Ende negiert sie das Ergebnis von W.
Nun geben wir der konstruierten Turing-Maschine M ihre eigne Repräsentation als Eingabe, d. h. $x = \langle M \rangle$ und überlegen uns, wie sie sich dann verhält:

- M hält auf der Eingabe $\langle M \rangle$. Denn das Duplizieren der Eingabe terminiert und W hält gemäß der Widerspruchsannahme auf jeder Eingabe.

- Wenn M auf der Eingabe $\langle M \rangle$ mit der Antwort "ja" hält, dann muß W auf der Eingabe $\langle M \rangle \# \langle M \rangle$ mit der Antwort "nein" gehalten haben, weil M das Resultat von W negiert. Gemäß Widerspruchsannahme gilt dann $\langle M \rangle \# \langle M \rangle \notin L_W$; also gilt gemäß Definition von L_W auch $\langle M \rangle \notin L(M)$, das heißt M hält auf der Eingabe $\langle M \rangle$ *nicht* mit der Antwort "ja".

  Wir haben also gezeigt, daß die Antwort „ja" nicht möglich ist. Bleibt also noch die zeite Möglichkeit der Antwort „nein".

- Wenn M auf der Eingabe $\langle M \rangle$ mit der Antwort "nein" hält, dann muß W auf der Eingabe $\langle M \rangle \# \langle M \rangle$ mit der Antwort "ja" gehalten haben. Gemäß Widerspruchsannahme gilt dann $\langle M \rangle \# \langle M \rangle \in L_W$; also gilt gemäß Definition von L_W auch $\langle M \rangle \in L(M)$, das heißt M hält auf der Eingabe $\langle M \rangle$ mit der Antwort "ja" (und *nicht* mit der Antwort "nein").

  Wir haben also gezeigt, daß auch die Antwort „nein" nicht möglich ist.

Weder die Antwort "ja" noch die Antwort "nein" ist also bei Eingabe $x = \langle M \rangle$ möglich. Da die Turing-Maschine aber per Konstruktion (und der Annahmen über W) immer terminiert, ist dies unmöglich. Da die einzige Annahme bei der Konstruktion von M die Existenz von W war, kann also auch W nicht existieren. Damit ist das Wortproblem (für Turing-Maschinen) nicht entscheidbar. □

Auch wenn man es auf den ersten Blick vielleicht nicht sieht, ist das Argument des obigen Beweises ein Diagonalisierungsargument. Um dies zu verdeutlichen, schreiben wir der Reihe nach die Repräsentationen aller Turingmaschinen als Zeilen- und Spaltenköpfe in eine Tabelle. Die Tabelle enthält die Ergebnisse von W bei der Eingabe von $\langle M_i \rangle \# \langle M_j \rangle$. Die Ergebnisse der Turingmaschine M bei Eingabe $\langle M_i \rangle$ ergeben sich dann dadurch, daß man die Einträge $\langle M_i \rangle \# \langle M_i \rangle$ der Diagonale negiert. Der Widerspruch enststeht im Schnittpunkt der Zeile und der Spalte $\langle M \rangle$: Wenn dort ein „ja" steht, muß dort (wegen der Negation der Diagonalen) ein „nein" stehen; wenn dort ein „nein" steht, muß dort ein „ja" stehen – unmöglich.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\ldots$	$\langle M \rangle$	$\ldots$
$\langle M_1 \rangle$	ja	ja	nein	ja		nein	
$\langle M_2 \rangle$	nein	ja	ja	nein		nein	
$\langle M_3 \rangle$	ja	nein	nein	ja		ja	
$\langle M_4 \rangle$	ja	nein	nein	ja		nein	
$\vdots$					$\ddots$		
$\langle M \rangle$	nein	nein	ja	nein		???	
$\vdots$							$\ddots$

6.3 Unentscheidbarkeit des Halteproblems

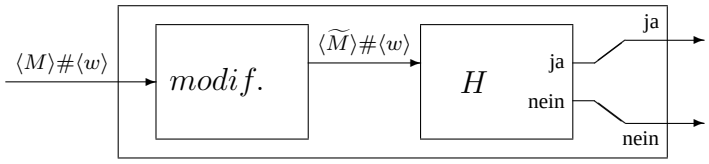
Mit fast demselben Argument könnte man auch zeigen, dass das Halteproblem nicht entscheidbar ist. Wir wenden jedoch eine andere Technik an, um die Unentscheidbarkeit des Halteproblems zu beweisen: Wir *reduzieren* das Wortproblem auf das Halteproblem. D. h. wir zeigen, daß man aus einer Maschine, die das Halteproblem entscheidet eine Maschine bauen könnte, die auch das Wortproblem entscheidet. Wenn nun das Halteproblem entscheidbar wäre, dann wäre damit auch das Wortproblem entscheidbar.

Da Diagonalisierungsargumente meist einen „Knoten im Hirn“ erzeugen, muß man bei ihnen viel Sorgfalt walten lassen. Deshalb sollte man, sich dieser Tortur nur einmal zu unterziehen. Danach ist meist die Technik der Reduktion die einfachere Beweistechnik zum Nachweis der Unentscheidbarkeit eines Problems. Diese Technik führen wir anhand eines Beispiels vor.

Folgerung 5.10 (Unentscheidbarkeit des Halteproblems)
Das Halteproblem L_H ist nicht entscheidbar.

Beweis: Angenommen es gäbe eine Turingmaschine H, die das Halteproblem entscheidet. Dann konstruieren wir daraus eine Turing-Maschine, die das Wortproblem entscheidet (wir reduzieren also das Wortproblem auf das Halteproblem):

- Eine deterministische Turingmaschine M kann man leicht so umbauen, dass sie ein Wort genau dann akzeptiert, wenn sie auf dem Wort hält (diese Turing-Maschine bezeichnen wir mit $\widetilde{M}$):
  - Lösche alle Übergänge $((q_e, a), (p, b, m))$ für $q_e \in F$.
  - Füge einen neuen Zustand q_l und Übergänge $((q_l, x), (q_l, x, r))$ für alle x hinzu.
  - Für jeden Zustand $q \notin F$ und jedes Bandzeichen a für das es keinen Übergang gibt, füge einen Übergang $((q, a), (q_l, a, r))$ hinzu.
- Die folgende Turingmaschine entscheidet dann das Wortproblem (wobei modif. eine Turing-Maschine ist die M in $\widetilde{M}$ umbaut):



Da das Wortproblem nicht entscheidbar ist, muß die getroffene Annahme falsch gewesen sein. Die Annahme, daß H existiert ist also falsch. Die Turing-Maschine H existiert also nicht.

□

Bemerkung: In der Literatur wird häufig zunächst die Unentscheidbarkeit des Halteproblems bewiesen, und dann das Halteproblem auf das Wortproblem reduziert, um zu beweisen, daß auch das Wortproblem unentscheidbar ist.

In einer ruhigen Minute, können Sie sich ja diese Reduktion einmal überlegen.

Wir haben zunächst das Wortproblem betrachtet, weil es eine Fragestellung ist, die nur die von der Turing-Maschine akzeptierte Sprache betrifft. Im folgenden werden wir sehen, daß keine derartige Eigenschaft entscheidbar ist: *Alle*[2] Fragen, die sich auf die von der Turing-Maschine akzeptierte Sprache beziehen, sind unentscheidbar.
Kurz gesagt gilt ist also die traurige Wahrheit: Die meisten Probleme für Turing-Maschinen sind nicht entscheidbar.
Beispiele für solche Probleme sind:

$$L(M) = \varnothing?$$
$$|L(M)| \geq 5?$$
$$L(M) \text{ regulär}?$$
$$L(M) \text{ kontextfrei}?$$

Wir könnten jetzt beginnen, die Unentscheidbarkeit für jedes einzelne Problem zu beweisen (zum Beispiel durch Reduktion des Halteproblems auf das entsprechende Problem). Der *1. Satz von Rice* spart uns diese Mühe, da er diese Aussage für alle Probleme, die sich auf die akzeptiertee Sprache einer Turing-Maschine beziehen, formuliert.

Der 1. Satz von Rice bzw. sein Beweis ist also ein Beweisschema, das für alle diese Probleme gleich funktioniert. Es ist immer dieselbe Reduktion.

Dazu müssen wir zunächst den Begriff der Eingeschaft einer Sprache und der nicht-trivialen Spracheigenschaften klären. Eine *Eigenschaft* von Sprachen ist eine Teilmenge P von aufzählbaren Sprachen (über einem geeigneten Alphabet). Wir schreiben $P(L)$, wenn L die Eigenschaft P erfüllt (d. h. wenn gilt $L \in P$). Wir schreiben $\neg P(L)$, wenn L die Eigenschaft P nicht erfüllt (d. h. wenn gilt $L \notin P$). Eine Eigenschaft P heißt *nicht-trivial*, wenn es wenigstens eine aufzählbare Sprache L mit $P(L)$ und wenigstens eine aufzählbare Sprache L' mit $\neg P(L')$ gibt.

Satz 5.11 (1. Satz von Rice)
Jede nicht-triviale Eigenschaft P (aufzählbarer Sprachen) ist unentscheidbar, d. h. die Sprache

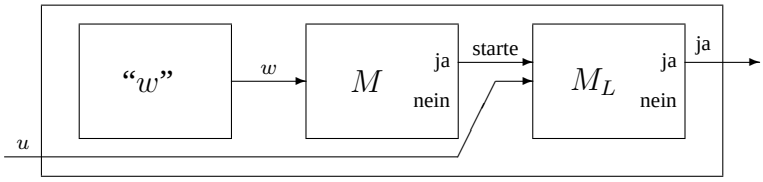
$$L_P = \{\langle M \rangle \mid M \text{ ist eine Turing-Maschine mit } P(L(M))\}$$

ist nicht entscheidbar.

Beweis: Ohne Beschränkung der Allgemeinheit können wir annehmen, dass $\neg P(\varnothing)$ gilt[3]. Da P per Voraussetzung nicht-trivial ist, gibt es auch eine aufzählbare Sprache L mit $P(L)$. Sei M_L eine Turing-Maschine mit $L(M_L) = L$. Wir beweisen nun durch Widerspruch, dass L_P nicht entscheidbar ist. Wir nehmen also an, daß L_P entscheidbar ist. Dann zeigen wir, daß damit das Wortproblem entscheidbar ist. Dazu gehen wir wie folgt vor:
Für eine Turing-Maschine M und eine Eingabe w konstruieren wir zunächst die folgende Turing-Maschine M_w:

[2] Ausnahmen sind lediglich die trivialen Eigenschaften.

[3] Wenn $P(\varnothing)$ nicht gilt, dann betrachten wir stattdessen das Komplement $\overline{P}$. Da die entscheidbaren Probleme unter Komplement abgeschlossen sind, ist P genau dann entscheidbar, wenn $\overline{P}$ entscheidbar ist.



M_w schreibt das Wort w auf das Band und startet dann die Turing-Maschine M auf diesem Band. Wenn M die Eingabe w akzeptiert, wird M_L auf der ursprünglichen Eingabe u gestartet. Das Ergebnis dieser Berechnung wird ausgegeben. Für Eingabe $\langle M \rangle \# \langle w \rangle$ lässt sich die Maschine $\langle M_w \rangle$ automatisch konstruieren.

Nun überlegen wir uns, welche Sprache M_w akzeptiert:

- Wenn $w \notin L(M)$ gilt (d. h. wenn die Turing-Maschine M das Wort w nicht akzeptiert), dann akzeptiert M_w keine Eingabe u, da M_L nie gestartet wird. Also gilt $L(M_w) = \varnothing$ und damit $\neg P(L(M_W))$.

- Wenn $w \in L(M)$ gilt, dann akzeptiert M_w genau die Wörter, die von M_L akzeptiert werden. Also gilt $L(M_w) = L$ und damit gilt $P(L(M_w))$

Wenn nun – wie angenommen – L_P entscheidbar ist, können wir das Wortproblem wie folgt entscheiden:

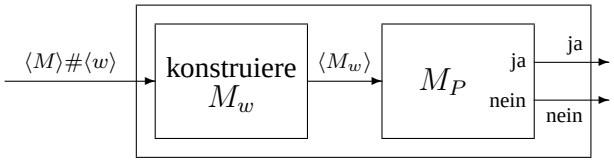
- Wir konstruiere aus M und w die Turing-Maschine M_w.

- Dann entscheiden wir, ob $P(L(M_w))$ gilt:

  · Wenn die Antwort „ja“ lautet, dann gilt $w \in L(M)$.

  · Wenn die Antwort „nein“ lautet, dann gilt $w \notin L(M)$

Dieses Entscheidungsverfahren ist nachfolgend schematisch als Turing-Maschine dargestellt. Dabei ist M_P die Turing-Maschine, die L_P entscheidet und „konstruiere M_w“ eine Turing-Maschine, die aus $\langle M \rangle \# \langle w \rangle$ die Repräsentation $\langle M_w \rangle$ konstruiert:



Diese Turing-Maschine würde das Wortproblem entscheiden, wenn denn die Maschine M_P existieren würde. □

Ein ähnlicher Satz gilt auch für aufzählbare Sprachen; das ist der 2. Satz von Rice, den wir in der Vorlesung jedoch nicht mehr behandeln können.

Beispiel 5.2 (Anwendung des 1. Satzes von Rice)
Hier sind einige Beispiele von Eigenschaften, für die der 1. Satz von Rice unmittelbar die Unentscheidbarkeit impliziert.

1. Es ist nicht entscheidbar, ob eine Turing-Machine mehr als sieben Wörter akzeptiert.

   Die Eigenschaft P ist also definiert durch $P(L) = \{L \mid |L| \geq 7\}$. Um den 1. Satz von Rice anwenden zu können, müssen wir noch überprüfen, ob die Eigenschaft P nicht-trivial ist. Wir müssen also mindestens eine aufzählbare Sprache finden, die die Eigenschaft erfüllt, und eine, die die Eigenschaft nicht erfüllt: Wir wählen die beiden Sprachen $\varnothing$ und $\{a, aa, aaa, \ldots, aaaaaaaa\}$. Offensichtlich gilt: $\neg P(\varnothing)$ und $P(\{a, aa, aaa, \ldots, aaaaaaaaaa\})$.

2. Es ist nicht entscheidbar, ob eine Turing-Maschine nur endlich viele Wörter akzeptiert (d. h. ob die von einer Turing-Maschine akzeptierte Sprache endlich ist).

   Die Eigenschaft P ist definiert durch $P(L) = \{L \mid |L| < \omega\}$. Die Eigenschaft ist nicht-trivial, da $\neg P(\Sigma^*)$ und $P(\varnothing)$ gilt.

3. Es ist nicht entscheidbar, ob die von einer Turing-Maschine akzeptierte Sprache regulär ist.

   Die Eigenschaft P ist definiert durch $P(L) = \{L \mid L \text{ ist regulär}\}$; sie ist nicht-trivial: $\neg P(a^n b^n)$ und $P(\varnothing)$.

Es gibt aber durchaus Eigenschaften für Turing-Maschinen, die entscheidbar sind. Für diese Eigenschaften ist dann aber eine der Voraussetzungen des 1. Satzes von Rice nicht erfüllt. Entweder handelt es sich nicht um eine Eigenschaft der akzeptierten Sprache der Turing-Maschine oder die Eigenschaft ist trivial

Beispiel 5.3 (Nicht-Anwendbarkeit des 1. Satzes von Rice)

1. Es ist entscheidbar, ob eine Turing-Maschine mindestens 7 Zustände besitzt.

   Diese Eigenschaft ist keine Eigenschaft der akzeptierten Sprache, sondern eine Eigenschaft der Turing-Maschine selbst.

2. Für ein Wort w ist entscheidbar, ob eine Turing-Maschine das Wort w nach höchstens 7 Schritten akzeptiert.

   Diese Eigenschaft ist keine reine Spracheigenschaft.

3. Es ist entscheidbar, ob die von einer Turing-Maschine akzeptierte Sprache aufzählbar ist.

   Diese Eigenschaft ist trivial (sie ist für jede aufzählbare Sprache per Definition erfüllt).

> **Frage:** *Ist entscheidbar, ob die von einer Turing-Maschine akzeptierte Sprache entscheidbar ist? Die Antwort darauf gibt es in Übung 13 (Aufgabe 1d).*

Achtung: *Wenn die Voraussetzungen des 1. Satzes von Rice verletzt sind, heißt das natürlich noch nicht, daß die Eigenschaft entscheidbar ist (das ist nur die notwendige Voraussetzung). Ein Beispiel dafür ist das Halteproblem.*

Den Beweis von Satz 3.31 hatten wir in Abschnitt 4.2 von Kapitel 3 zurückgestellt. Jetzt sind wir in der Lage, diesen Beweis zu führen. Allerdings belassen wir es hier bei der Idee. Der Beweis wird dann in der Übung (Blatt 11, Aufgabe 2 und Blatt 12, Aufgabe 1.c) geführt.

Satz 5.12 (Satz 3.31)
Es ist nicht entscheidbar, ob die von zwei kontextfreien Grammatiken erzeugten Sprachen disjunkt sind.

Beweis: Idee: Die akzeptierenden Berechnungen einer Turing-Maschine M für ein Wort w lassen sich als Durchschnitt zweier kontextfreier Sprachen darstellen.
Wenn wir die Disjunkheit dieser kontextfreien Grammatiken entscheiden könnten, könnten wir damit das Wortproblem entscheiden: Sind die Grammatiken disjunkt, akzeptiert M das Wort w nicht. Sind sie nicht disjunkt, akzeptiert M das Wort w.
(Details hierzu werden in den Übungen Blatt 11 und 12 behandelt). □

Bemerkung: Es gibt noch ein weiteres sehr interessantes unentscheidbares Problem: Das *Post'sche Korrespondenz Problem* (PKP/PCP). Leider fehlt uns die Zeit, dieses Problem genauer zu betrachten. Das sollten Sie sich aber unbedingt in der Literatur ansehen. Der Beweis der Unentscheidbarkeit des PKPs ähnelt dem Beweis der Unentscheidbarkeit des Disjunktheitsproblems für kontextfreie Grammatiken.

7 Aufzählbare und nicht-aufzählbare Probleme

Wir hatten uns bereits die Frage gestellt, ob es Probleme gibt, die aufzählbar, aber nicht entscheidbar sind (nur dann ist die Unterscheidung der Begriffe sinnvoll). In diesem Abschnitt werden wir zwei solche Probleme angeben:

- Das Wortproblem
- Das Halteproblem

Als Konsequenz ergibt sich dann sofort, daß die Komplemente dieser Probleme nicht aufzählbar sind (denn sonst wären sie entscheidbar).
Um das Wortproblem oder das Halteproblem aufzuzählen, müssen wir im wesentlichen eine gegebene Turing-Maschine M auf einem gegebenem Wort w simulieren. Wir benötigen also ein Turing-Maschine, die eine andere gegebene Turing-Maschine M auf einem Wort w simuliert: die *universelle Turing-Maschine*.

Die universelle Turing-Maschine U Es gibt eine Turing-Maschine U, die als Eingabe eine Turing-Maschine M und ein Wort w benötigt (codiert als $\langle M\rangle\#\langle w\rangle$); U simuliert dann die Berechnung von M auf w. Wenn M das Wort w akzeptiert, dann akzeptiert U das Wort $\langle M\rangle\#\langle w\rangle$; d. h. $L(U) = L_W$. U heißt *universelle Turing-Maschine* und L_W wird auch die *universelle Sprache* genannt.
Wir geben hier die universelle Turing-Maschine U nicht explizit an. Wir überlegen uns nur ganz grob, wie sie aufgebaut sein könnte (und damit, daß sie existiert). Die universelle Turing-Maschine hat drei Bänder. Auf dem ersten Band steht die Eingabe $\langle M\rangle\#\langle w\rangle$, auf dem zweiten Band wird das Band von M simuliert und auf dem dritten ist der aktuelle Zustande von M codiert. Am Anfang wird das Wort w aus der Eingabe auf das zweite Band kopiert und der Anfangszustand (bzw. eine Kodierung davon) auf das dritte Band geschrieben. Danach werden die Übergänge der auf dem ersten Band kodierten Maschine M simuliert.
Mit $L(U) = L_W$ ist damit das Wortproblem offensichtlich aufzählbar.

Satz 5.13 (Aufzählbarkeit des Wortproblems)
Das Wortproblem L_W ist aufzählbar.

Beweis: Die universelle Turing-Maschine (die wir streng genommen noch explizit angeben müßten) akzeptiert genau die Sprache L_W. Damit ist L_W aufzählbar. □

Folgerung 5.14

1. *Das Halteproblem L_H ist aufzählbar.*

   Beweisidee: *Reduktion des Halteproblems auf das Wortproblem.*

2. *Das Komplement des Wortproblems $\overline{L_W}$ und das Komplement des Halteproblems $\overline{L_H}$ sind* nicht *aufzählbar.*

   Beweisidee: *Sonst wären L_W und L_H gemäß Satz 5.8 entscheidbar. Dies ist ein Widerspruch zu Satz 5.9 bzw. Folgerung 5.10 (der Unentscheibarkeit von L_W bzw. L_H).*

Spannende Frage: *Warum funktioniert das Diagonalisierungsargument aus dem Beweis der Unentscheidbarkeit von L_W (Satz 5.9) nicht auch für die Aufzählbarkeit?*
Antwort: *Aus einer Nicht-Terminierung, die einem "nein" entspricht, kann man kein "ja" machen! "nein" bleibt damit "nein". Also ergibt sich im "nein"-Fall kein Widerspruch.*

Bemerkungen: Alle Sprachen, die sich durch ein *endliches* Regelsystem beschreiben lassen, sind aufzählbar.

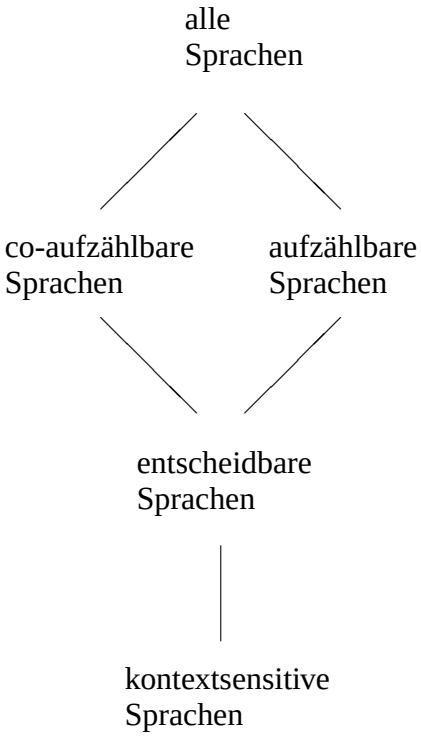
- Eine Sprache, die durch eine *beliebige* Grammatik G erzeugt wird, ist aufzählbar. Die von Typ-0-Grammatiken erzeugten Sprachen sind genau die aufzählbaren Sprachen.

- Die Sprache der in einem logischen Kalkül herleitbaren Formeln bzw. Ausdrücke ist aufzählbar.

- Insbesondere ist die Sprache aller allgemeingültigen Formeln der Prädikatenlogik 1. Stufe (PL1) aufzählbar (aber *nicht* entscheidbar).

8 Überblick

Die nachfolgende Graphik gibt nochmal einen Überblick über die in diesem Kapitel diskutierten Sprachklassen. Dabei ist eine Sprache co-aufzählbar, wenn ihr Komplement (engl. complement) aufzählbar ist. Eine Sprache ist also genau dann entscheidbar, wenn sie sowohl aufzählbar als auch co-aufzählbar ist.


alle
Sprachen
co-aufzählbare
Sprachen
aufzählbare
Sprachen
entscheidbare
Sprachen
kontextsensitive
Sprachen

Kapitel 6

Berechenbare Funktionen

1 Motivation und Definition

Bisher haben wir Turing-Maschinen bzw. Automaten zum Erkennen bzw. Akzeptieren von Sprachen benutzt. Die Eingabe ist dabei ein Wort über einem bestimmten Alphabet; die Ausgabe ist – wenn die Maschine auf der Eingabe terminiert – eine Antwort "ja" oder "nein". Im folgenden benutzen wir die Turing-Maschinen zum Berechnen von Funktionen – wie bereits bei der Definition der Turing-Maschinen im vorangegangenen Kapitel motiviert.
Traditionell betrachtet man bei der Untersuchung des Berechnungsbegriffs nur Funktionen über den natürlichen Zahlen.

$$f : \mathbb{N}^n \to \mathbb{N}$$

Die Funktionen können jedoch *partiell*[1] sein, d. h. f ist nicht für jedes $\underline{x} = (x_1, \ldots, x_n)$ definiert, was oft durch $f(\underline{x}) = \perp$ oder $f(\underline{x}) \notin \mathbb{N}$ notiert wird.

Idee Wir funktionieren also zunächst die Turing-Maschinen so um, daß wir ihre Berechnung als die Berechnung einer Funktion auffassen können. Dazu kodieren wir die Eingabe $\underline{x} = (x_1, \ldots, x_n)$ durch $\#bin(x_1)\#bin(x_2)\# \ldots \#bin(x_n)\#$ auf dem Eingabeband einer Turing-Maschine, wobei $bin(x_i)$ die Binärdarstellung der Zahl x_i ist. Am Ende soll dann das Ergebnis $f(x)$ der Berechnung binär kodiert (also $bin(f(\underline{x}))$) auf dem Band stehen.

> *Die Kodierung der Zahlendarstellungen variieren. Teilweise werden Zahlen auch unär kodiert oder andere Trennzeichen (z. B. das Blank) benutzt. Für unsere prinzipiellen Überlegungen spielt die konkrete Kodierung keine Rolle (bei Komplexitätsbetrachtungen dagegen schon).*

Wir können damit nun die berechenbaren Funktionen – genauer die Turing-berechenbaren Funktionen – definieren .

Definition 6.1 (Berechenbare Funktionen) *Eine Funktion $f : \mathbb{N}^n \to \mathbb{N}$ heißt (Turing)*-berechenbar, *wenn es eine Turing-Maschine M gibt, die*

[1] Wir werden später sogar sehen, daß jeder hinreichend mächtige Berechnungsbegriff die Definition von partiellen Funktionen zulassen muß.

- *für jedes* $\underline{x} = (x_1, \ldots, x_n) \in \mathbb{N}^n$ *mit* $f(\underline{x}) \in \mathbb{N}$ *auf der Eingabe*

$$\#bin(x_1)\#bin(x_2)\#\ldots\#bin(x_n)\#$$

mit Bandinhalt $\#bin(f(x_1,\ldots,x_n))\#$ *hält und*

- *für jedes* $\underline{x} = (x_1, \ldots, x_n) \in \mathbb{N}^n$ *mit* $f(\underline{x}) = \perp$ *auf der Eingabe*

$$\#bin(x_1)\#bin(x_2)\#\ldots\#bin(x_n)\#$$

nicht hält.

Achtung: *Hier gibt es eine ganze Reihe technisch verschiedener aber konzeptionell äquivalenter Definitionen. Einige Definitionen sind sogar konzeptionell verschieden; sie führen alle zum selben Berechnungsbegriff. Der Grund dafür ist wieder die Church'che These.*

Bemerkung:

- Eine berechenbare Funktion wird auch *partiell-rekursiv* genannt.
- Eine berechenbare Funktion, die total ist (d. h. $f(\underline{x}) \in \mathbb{N}$ für alle $\underline{x} \in \mathbb{N}^n$) wird auch *total-rekursiv* genannt.

Beispiel 6.1

- Jede Funktion, die Sie in Ihrer Lieblingsprogrammiersprache formulieren können, ist berechenbar: Ein beliebtes Beispiel ist die Ackermannfunktion[2].

  Hier ist eine mögliche Formulierung der Ackermannfunktion $a : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$:

  · $a(0,0) = 1$
  · $a(0,1) = 2$
  · $a(0,y) = 2 + y$ für $x \geq 2$
  · $a(x+1,0) = 1$
  · $a(x+1,y+1) = a(x, a(x+1,y))$

  Die Ackermannfunktion ist ein Funktion, die extrem schnell wächst – schneller als alles, was wir uns normalerweise vorstellen können (denn Herr Ackermann hat sie extra so definiert). Hier ein paar Anhaltspunkte dafür:

  · $a(0,y) = 2 + y$ für $y \geq 2$ (Addition für $x = 0$)
  · $a(1,y) = 2 * y$ für $y \geq 1$ (Multiplikation für $x = 1$)
  · $a(2,y) = 2^y$ für $y \geq 0$ (Potenzfunktion für $x = 2$)
  · $a(3,y) = 2^{2^{\vdots^{2}}}$ für $y \geq 1$ (y-malige Anwendung der Potenz für $x = 3$)

[2] Eine Grund für ihre Beliebtheit werden wir später noch kennenlernen: Sie ist ein Beispiel für eine nicht primitiv rekursive Funktion, die total rekursiv ist.

· ...

Wenn x um eins erhöht wird, „potenziert“ sich die auf y angewendete Funktion: Addition $\rightarrow$ Multiplikation $\rightarrow$ Potenz $\rightarrow$ Hyperpotenz $\rightarrow$ .... Bereits für $x = 4$ gibt es keine Notation oder Bezeichnung mehr für $a(4, y)$.

- Gegenbeispiel: Die charakteristische Funktion des Halteproblems ist nicht berechenbar. Genauer sei M_1, M_2, M_3, ... eine Numerierung aller Turing-Maschinen:

$$f : \mathbb{N} \rightarrow \mathbb{N} \quad f(i) = \begin{cases} 1 & \text{falls } M_i \text{ mit leerer Eingabe hält} \\ 0 & \text{sonst} \end{cases}$$

2 Registermaschinen

Es ist sehr mühselig, Turing-Maschinen „zu programmieren“. Im folgenden Abschnitt betrachten wir nun ein Berechnungsmodell, das der Vorstellung eines (hardware-nahen) Informatikers vom Programmieren besser entspricht: die sog. *Registermaschine* (engl. random access machine, kurz RAM genannt).
Wir nennen die Funktionen, die man mit Registermaschinen berechnen kann, *registermaschinenberechenbar*. Obwohl dieser Begriff konzeptuell und technisch verschieden definiert ist, werden wir sehen, daß er genau die Turing-berechenbaren Funktionen charakterisiert.

2.1 Das Konzept der Registermaschine

Zunächts stellen wir das Konzept der Registermaschine vor.

- Die Registermaschine besitzt unendlich viele *Register* R_0, R_1, R_2, ...; jedes Register kann eine natürliche Zahl speichern!

  Der Index i eines Registers i wird oft auch die Adresse des Registers genannt. Die Register entsprechen im wesentlichen dem unendlichen Band der Turing-Maschine.

- R_0 ist ein ausgezeichnetes Register, das auch *Akkumulator* genannt wird.

- Initial enthalten die Register R_1, ... , R_n die Werte $x_1, \ldots, x_n$. Alle anderen Register enthalten den Wert 0.

- Am Ende der Berechnung steht das Ergebnis der Berechnung im Akkumulator.

- Die Berechnung selbst wird durch ein Progamm beschrieben; das Programm besteht aus einer durchnumerierten endlichen Folge von Befehlszeilen.

- Ein *Befehlszähler* Z gibt die Nummer des nächsten auszuführenden Befehls an.

  Das Programm nebst Befehlszähler entspricht der endlichen Kontrolle der Turing-Maschine.

- Der genaue Befehlssatz der verschiedenen Registermaschinenmodelle variiert. Hier geben wir einen möglichem Befehlssatz an:

Befehl		Semantik
LDA	i	$R_0 := R_i; Z := Z + 1$
STA	i	$R_i := R_0; Z := Z + 1$
ADD	i	$R_0 := R_0 + R_i; Z := Z + 1$
SUB	i	$R_0 := R_0 - R_i; Z := Z + 1$
INC	i	$R_i := R_i + 1; Z := Z + 1$
DEC	i	$R_i := R_i - 1; Z := Z + 1$
JMP	i	$Z := i$
JMP	i	$\begin{cases} R_0 = 0 : Z := i \\ R_0 \neq 0 : Z := Z + 1 \end{cases}$
END		$Z :=$ Letzter Befehl $+ 1$

Oft sieht man Registermaschinen, die auch Befehle zur Multilikation oder Division besitzen. Diese Befehle kann man aber durch Additionen bzw. Subtraktionen simulieren. Wir könnten sogar auf die Addition und Subtraktion verzichten, denn diese könnte man durch Inkrementierung und Dekrementierung simulieren.

- Das Programm *hält*, wenn der Befehlszähler auf eine nicht existierende Befehlszeile verweist.

Die obige Definition ist zwar nur „semi-formal“, reicht aber für unsere Zwecke aus. Wir müßten für die folgende Definition die Berechnungen einer Registermaschine noch genauer formalisieren – das ist nicht kompliziert aber etwas mühselig. Wir sparen uns (aus Zeitgründen) die Mühe.

Definition 6.2 (Registermaschinenberechenbarkeit)
Eine (partielle) Fuktion heißt registermaschinenberechenbar, *wenn es ein Registermaschinenprogramm gibt, das diese Funktion berechnet.*

Beispiel 6.2
Wir betrachten die Funktion $f(x) = x^2$, die sich mit dem folgenden Programm berechnen lässt. Dabei steht der Parameter x im Register 1; das Register 2 nutzen wir als Zähler z; das Register 3 ist siw Ergebnisvariable erg:

1	LDA	1	x in den Akku
2	STA	2	$z := x$ (und z in Akku für die nachf. Bed.)
3	JPZ	10	Wenn $z = 0$ dann goto 10
4	DEC	2	$z := z - 1$
5	LDA	3	erg:= erg + x
6	ADD	1	|
7	STA	3	|
8	LDA	2	z in Akku (für Bed.)
9	JMP	3	goto 3
10	LDA	3	erg nach Akku

2.2 Vergleich der Berechenbarkeitsbegriffe

Als nächstes vergleichen wir den Begriff der Turing-berechenbaren Funktionen mit dem Begriff der registermaschinenberechenbaren Funktionen. Zunächst zeigen wir, daß wir alles was eine Registermaschine berechnen kann auch mit einer Turing-Maschine berechnen können.

Satz 6.3 *Jede registermaschinenberechenbare Funktion ist (Turing-) berechenbar.*

Beweis: Simuliere die Registermaschine durch eine Turing-Maschine, wobei das Band die Register R_0, R_1, R_2, ... repräsentiert.

$$\#bin(R_0)\#bin(R_1)\#bin(R_2)\#\ldots$$

Der aktuelle Befehlszähler kann im Zustand der Turing-Maschine gespeichert werden (da es ja nur endlich viele Befehlszeilen gibt). Die konkreten Befehle können durch entsprechende Makros realisiert werden. Dies müssten wir uns für jeden einzelnen Befehl des Befehlssatzes der Registermaschine überlegen; das ist nicht schwer – aber aufwendig. □

Als nächstes betrachten wir die umgekehrte Richtung. Wir überlegen uns dazu, daß wir jeden Berechnungsschritt einer Turing-Maschine durch eine Registermaschine simulieren können. Den unendlichen Bandinhalt der Turing-Maschine repräsentieren wir dazu in zwei Registern (eines repräsentiert das Band links vom Kopf, das zweite repräsentiert das Band rechts vom Kopf).

Vorüberlegung: Dazu müssen wir ein paar Überlegungen anstellen: Es gibt registermaschinenberechenbare Funktionen:

$$\begin{aligned} cons: &\quad \mathbb{N}\times\mathbb{N}\to\mathbb{N} \\ first: &\quad \mathbb{N}\to\mathbb{N} \\ rest: &\quad \mathbb{N}\to\mathbb{N} \end{aligned}$$

mit

$$\begin{aligned} &cons(0,0)=0 \\ &first(cons(x,y))=x \\ &rest(cons(x,y))=y \end{aligned}$$

Dabei interpretieren wir den ersten Parameter von $cons$ als ein Zeichen und den zweiten Parameter als eine (kodierte) Zeichenreihe. Die Funktion $cons(x,y)$ können wir dann als das Anfügen eines Zeichens x an die Zeichenreihe y auffassen. Die Funktion $first$ spaltet wieder das erste Zeichen ab; die Funktion $rest$ liefert die Kodierung des verbleibenden Teils der Zeichenreihe.
Das Blank wird durch $x=0$ und alle anderen Zeichen des Bandalphabetes werden beginnend mit 1 durchnumeriert. Das leere (nur mit Blanks beschriftete) Band wird ebenfalls durch 0 repräsentiert. Die Bedingung $cons(0,0)=0$ garantiert dann, daß das Anfügen eines Blanks an ein leeres Band wieder das Blank liefert. Umgekehrt liefert $first(0)$ das Blank und $rest(0)$ das leere Band, was mit unserer Interpretation genau übereinstimmt.

Beispielsweise könnten wir $cons(x,y)=\frac{(x+y)(x+y+1)}{2}+x$ definieren (vgl. Übung 1, Aufgabe 4g.).

Redaktioneller Hinweis: Hier sollte irgendwann nochmal das Bild aus dem handschriftlichen Skript (S. 175) hin.

Die Repräsentation von Zeichenreihen durch natürliche Zahlen nennt man auch Gödelisierung. *Denn Herr Gödel hat diesen Trick (mit einer anderen Kodierung) benutzt, um in der Arithmetik Aussagen über die Arithmethik zu formulieren, was zu seinen berühmten Unvollständikeitssätzen geführt hat. Die Konsequenzen unserer Gödelisierung sind dagegen ziemlich harmlos.*

Mit diesen Funktionen können wir nun die Konfigurationen einer Turing-Maschine den Übergang in die Nachfolgekonfiguration simulieren. Eine Konfiguration $a_k a_{k+1} \ldots a_1 q_j b_1 b_2 \ldots b_l$ einer Turing-Maschine repräsentieren wir in drei Registern:

$$\begin{array}{lcl} R_1 & := & cons(a_1, cons(a_2, cons(a_3, \ \ldots \ cons(a_k, 0) \ \ldots))) \\ R_2 & := & j \\ R_3 & := & cons(b_1, cons(b_2, cons(b_3, \ \ldots \ cons(b_l, 0) \ \ldots))) \end{array}$$

Die Nummer des Zeichens an der Kopfposition ist dann $b_1 = first(R_3)$.
Die Nachfolgekonfiguration können wir ebenfalls einfach berechnen. Betrachten wir als Beispiel eine Rechtsbewegung einer Turing-Maschine M:

$$\begin{array}{c} a_k a_{k+1} \ldots a_1 q_j b_1 b_2 \ldots b_l \\ \vdash_M \\ a_k a_{k+1} \ldots a_1 b_1' q_j' b_2 \ldots b_l \end{array}$$

Das Zeichen b_1' erhalten wir unmittelbar aus der Übergangsfunktion δ der zu simulierenden Turing-Maschine (die wir im Registermaschinenprogramm als Entscheidungstabelle realisieren können), indem wir den Wert für $first(R_3)$ und j ermitteln. Auch j' erhalten wir direkt aus der Übergangsfunktion δ. Damit ergibt sich der Inhalt der Register für die Nachfolgekonfiguration durch:

$$\begin{array}{lcl} R_1 & := & cons(b_1', R_1) \\ R_2 & := & j' \\ R_3 & := & rest(R_3) \end{array}$$

Für eine Linksbewegung ergibt sich die Nachfolgekonfiguration analog.

Satz 6.4 *Jede (Turing-)berechenbare Funktion ist registermaschinenberechenbar.*

Beweis: Simulation der Turing-Maschine durch die Registermaschine gemäß obiger Überlegung. □

Bemerkungen:

- Die Registermaschine ist für Komplexitätsbetrachtungen oft zweckmäßiger als eine TM.
- Man kann den Befehlssatz der Registermaschine auch noch erweitern: zum Beispiel um die Befehle MULT, DIV oder um Befehle mit *indirekter Adressierung*. Dadurch erhalten wir keine neuen berechenbaren Funktionen, erhöht aber die Programmierbarkeit der Registermaschine.

  Beweisidee: *Man kann jede Funktion, die sich mit einer Registermaschine mit indirekter Adressierung berechnen lässt, auch mit einer Turing-Maschine berechnen (Verallgemeinerung von Satz 6.3 auf Registermaschinen mit indirekter Adressierung); alle Funktionen, die sich mit einer Turing-Maschine berechnen lassen, lassen sich wiederum auch durch eine Registermaschine ohne indirekte Adressierung berechnen (Satz 6.4).*

- Oft werden die Konfigurationen durch die Primzahlcodierungen repräsentiert. In der Primzahlcodierung lässt sich die Ziffernfolge 1012134 mittels der Zahl $2^1 \cdot 3^0 \cdot 5^1 \cdot 7^2 \cdot 11^1 \cdot 13^3 \cdot 17^4$ darstellen. Hierbei wird die i-te Stelle durch die i-te Primzahl codiert und der Wert der Ziffer an dieser Stelle durch die Potenzierung der Primzahl.

3 GOTO-, WHILE- und FOR-Berechenbarkeit

Der Befehlssatz der Registermaschine sieht bisher nur Sprünge (JMP) und bedingte Sprünge (JPZ) als Konstrukte vor. Das wesentliche Konstrukt ist also die GOTO-Anweisung. Wir nennen diese Programme deshalb auch GOTO-Programme; und die Registermaschinenberechenbarkeit nennen wir auch GOTO-Berechenbarkeit.
GOTO-Anweisungen wird in der Programmierung jedoch als schlechter Stil angesehen, weil man mit ihnen sehr unübersichtliche Programme schreiben kann[3]. In der *Strukturierten Programmierung* beschränkt man sich deshalb auf eine kontrolliertere Art von Sprüngen; ein Programm wird dort aus den folgenden Kontrollkonstrukten aufgebaut:

- Sequenz: $S_1; S_2$
- Alternative: IF B THEN S_1 ELSE S_2 FI
  Hierbei ist B ein boolescher Ausdruck (bei uns ist dies ein beliebiges Register R_j, das implizit auf 0 getestet wird ($R_j == 0$)).
- Iteration:
  - WHILE-Schleife: WHILE B DO S OD
  - FOR-Schleife: FOR i DO S OD
    Mit i wird in der FOR-Schleife das Register R_i adressiert, das wir *Schleifenzähler* nennen.Der Wert des Registers wird bei jedem Schleifendurchlauf um 1 dekrementiert. Nimmt das Schleifenzähler den Wert 0 an, wird die FOR-Schleife beendet. Im Schleifenrumpf S darf der Schleifenzähler R_i dabei nicht modifiziert werden (reine) FOR-Schleife.

Im folgenden unterscheiden wir Programme, gemäß der in ihnen vorkommenden Schleifenkonstrukte: In WHILE-Programmen kommen nur WHILE-Schleifen vor; in FOR-Programmen kommen nur FOR-Schleifen vor. Entsprechend definieren wir dann den Begriff der WHILE- und FOR-Berechenbarkeit.

Definition 6.5 (WHILE- **und** FOR-**Berechenbarkeit)**

1. *Ein Registermaschinenprogramm, das nur aus elementaren Anweisungen (*STA, LDA, ADD, SUB, INC, DEC*) und aus Sequenzen, Alternativen und* WHILE*-Schleifen aufgebaut ist, heißt* WHILE *-Programm.*
2. *Ein Registermaschinen-Programm, das nur aus elementaren Anweisungen und aus Sequenzen, Alternativen und (reinen)* FOR*-Schleifen aufgebaut ist heißt* FOR*-Programm (manchmal auch* LOOP*-Programm).*

[3] E. W. Dijkstra: Go to statement considered harmful. CACM 11/3, 1968.

3. *Eine Funktion heißt* FOR*-berechenbar, wenn es ein* FOR*-Programm gibt, das diese Funktion berechnet.*

4. *Eine Funktion heißt* WHILE*-berechenbar, wenn es ein* WHILE*-Programm gibt, das diese Funktion berechnet.*

Nun überlegen wir uns, wie sich die neu definierten Berechenbarkeitsbegriffe zum bisherigen Berechenbarkeitsbegriff verhalten. Dazu stellen wir einige Beobachtungen an.

Beobachtungen:

- Jede FOR-Schleife kann natürlich durch eine WHILE-Schleife ersetzt werden. Deshalb ist jede FOR-berechenbare Funktion auch WHILE-berechenbar.

- Jedes FOR-Programm terminiert für alle Eingaben! D.h. alle FOR-berechenbaren Funktionen sind total.

  Beweis*: Falls das* FOR*-Programm nur eine Schleife besitzt, ist die Terminierung klar, da der Schleifenzähler in jedem Durchlauf genau um 1 verringert wird. Da der Schleifenzähler im Schleifenrumpf nicht verändert werden kann, erreicht er irgendwann den Wert 0. Dann ist die Schleife beendet. Für Programme mit ineinander verschachtelten Schleifen folgt die Aussage durch Induktion.*

  Wir werden in Übung 13, Aufgabe 2 sehen, daß in einer Programmiersprache, deren Programme immer terminieren nicht alle totalen und berechenbaren Funktionen berechnet werden können. Ein Beispiel für eine totale und berechenbare Funktion, die nicht FOR-berechenbar ist, ist die Ackermannfunktion.

- Jede WHILE-berechenbare Funktion kann durch ein WHILE-Programm berechnet werden, das genau eine WHILE-Schleife hat (dies nennt man dann ein Programm in *Kleene'scher Normalform*).

  Beweis: *Zunächst kann man natürlich ein* WHILE*-Programm in ein* GOTO*-Programm übersetzen. Dieses* GOTO*-Programm kann man dann in eine Turing-Maschine übersetzen (Satz 6.3). Die Turing-Maschine kann mann dann durch ein* WHILE*-Programm simulieren, wobei in der einen* WHILE*-Schleife genau ein Schritt der Turing-Maschine simuliert wird (so ähnlich wie in Satz 6.4).*
  Insgesamt haben wir dann über einige Umwege aus einem WHILE*-Programm mit evtl. mehreren* WHILE*-Schleifen ein äquivalentes* WHILE*-Programm mit genau einer* WHILE*-Schleife übersetzt.*

Insgesamt egibt sich dann das folgende Bild:

Satz 6.6

1. *Die* WHILE*-berechenbaren Funktionen sind genau die berechenbaren Funktionen.*

2. *Die* FOR*-berechenbaren Funktionen sind eine* echte *Teilmenge der berechenbaren Funktionen.*

Beweis:

- Wie in der vorangegangenen Beobachtung angedeutet, können wir jedes WHILE-Programm in ein äquivalentes GOTO-Programm übersetzen. Damit ist jede WHILE-berechenbare Funktion auch GOTO-berechenbar und damit (Turing-)berechnenbar.

  Umgekehrt kann man, wie oben angedeutet, eine Turing-Maschine durch ein WHILE-Programm simulieren. Damit ist jede (Turing-)berechenbare Funktion auch WHILE-berechenbar (das geht wie im Beweis von Satz 6.4).

- Obige Beobachtung und Übungsblatt 13, Aufgabe 2.

  Eine sehr interessante Aufgabe ist auch der Nachweis, daß die Ackermannfunktion nicht FOR*-berechenbar ist. Dazu zeigt man, daß ein FOR-Programm für die Berechnung von a(n+1,n+1) mindestens n ineinander verschachtelte* FOR*-Schleifen benötigt (das war genau die Absicht von Herrn Ackermann bei der Konstruktion seiner Funktion). Da dies für beliebige n gilt, gibt es keine Obergrenze für die Anzahl der verschachtelten* FOR*-Schleifen. Leider fehlt uns die Zeit, uns diesen Beweis genauer anzusehen.*

□

4 Primitive Rekursion und μ-Rekursion

Die bisherigen Definitionen der Berechnungsbegriffe entsprechen dem Prinzip der imperativen Programmierung. Als letztes betrachten wir nun das Prinzip der funktionalen Programmierung und definieren zwei entsprechende Berechenbarkeitsbegriffe.

Funktionale Programmierung: Gemäß des Prinzips der funktionalen Programmierung werden Funktionen durch Kombination von elementaren Funktionen definiert. Darüber hinaus können Funktionen rekursiv definiert werden.
Dabei müssen wir noch festlegen, welche Funktionen wir als elementar ansehen, wie wir aus bereits definierte Funktionen neue konstruieren können, und welche Form der Rekursion wir zulassen. Wir betrachten hier zwei verschiedene Formen der Rekursion: die primitive (oder begrenzte) Rekursion, und die (unbeschränkte) μ-Rekursion. Die elementaren Funktionen und die Kombination von Funktionen sind in beiden Fällen gleich

4.1 Primitive Rekursion

Zunächts definieren wir die primitiv rekursiven Funktionen.

Definition 6.7 (Primitiv rekursive Funktionen)
Die Menge der primitiv rekursive Funktionen *ist induktiv definiert durch:*

Elementare Funktionen *Die folgenden Funktionen sind primitiv rekursiv:*

Konstante Null $0 : \mathbb{N}^0 \to \mathbb{N}$ *mit* $0() = 0$.

Nachfolgerfunktion $\sigma : \mathbb{N} \to \mathbb{N}$ *mit* $\sigma(x) = x + 1$ *für alle* $x \in \mathbb{N}$.

Projektionsfunktionen $\pi_i^n : \mathbb{N}^n \to \mathbb{N}$ *mit* $\pi_i^n(x_1, ..., x_n) = x_i$ *(für* $1 \leq i \leq n$*).*

Konstruktionen

Einsetzen *Wenn* $g : \mathbb{N}^k \to \mathbb{N}$ *und* $h_i : \mathbb{N}^n \to \mathbb{N}$ *primitiv rekursive Funktionen sind, dann ist auch* $f : \mathbb{N}^n \to \mathbb{N}$ *mit* $f(x_1, ..., x_n) = g(h_1(\underline{x}), \ldots, h_k(\underline{x}))$ *primitiv rekursiv.*

Primitive Rekursion *Wenn* $g : \mathbb{N}^n \to \mathbb{N}$ *und* $h : \mathbb{N}^{n+2} \to \mathbb{N}$ *primitiv rekursive Funktionen sind, so ist auch* $f : \mathbb{N}^{n+1} \to \mathbb{N}$ *mit*
$f(\underline{x}, 0) = g(\underline{x})$ *Terminierung*
$f(\underline{x}, k+1) = h(\underline{x}, k, f(\underline{x}, k))$ *Rekursion*
ebenfalls primitiv rekursiv.

Ähnlich wie bei der (reinen) FOR-Schleife, ist bei bei der primitiven Rekursion die Anzahl der rekursiven Aufrufe begrenzt: für $f(x, n)$ wird f genau n mal rekursiv aufgerufen. Aller primitiv rekursiven Funktionen sind also tolal. Und man kann sich relativ leicht überlegen, daß die primitiv rekursiven Funktionen genau die FOR-berechenbaren Funktionen sind:

Satz 6.8 (Primitiv rekursiv = FOR**-berechenbar)** *Die Menge der primitiv rekursiven Funktionen ist genau die Menge der* FOR*-berechenbaren Funktionen.*

Frage: Welches Rekursionsprinzip liefert uns genau die WHILE-berechenbaren Funktionen?

Antwort:

4.2 Die μ-rekursiven Funktionen

Für eine Funktion: $g : \mathbb{N}^{n+1} \to \mathbb{N}$ bezeichnet $\mu g : \mathbb{N}^n \to \mathbb{N}$ eine Funktion mit:

$$(\mu g)(\underline{x}) = \begin{cases} \min\{k \in \mathbb{N} \mid g(\underline{x}, k) = 0 \wedge g(\underline{x}, j) \in \mathbb{N} \text{ für alle } j \leq k\} \\ \text{undefiniert, falls } \min \text{ nicht existiert} \end{cases}$$

$\mu g(\underline{x})$ bezeichnet also den kleinsten Index k, für den $g(\underline{x}, k) = 0$ gilt und für den für alle $j < k$ der Wert $g(\underline{x}, j)$ definiert ist.

Der μ-Operator macht also aus einer $n+1$-stelligen Funktion g eine n-stellige Funktion $f = \mu g$. Der μ-Operator ist also streng genommen ein Funktional $f = \mu(g)$, wobei die Klammern meist weggelassen werden – wie oben.

Definition 6.9 (μ-rekursive Funktionen)
Die Menge der μ-rekursive Funktionen ist induktiv definiert:

- *Jede primitiv rekursive Funktion ist μ-rekursiv.*
- *Wenn g und $h_1, ..., h_k$ μ-rekursiv sind, dann ist auch $f : \mathbb{N}^n \to \mathbb{N}$ mit $f(\underline{x}) = g(h_1(\underline{x}), \ldots, h_k(\underline{x}))$ μ-rekursiv.*
- *Wenn g μ-rekursiv ist, dann ist auch die Funktion μg μ-rekursiv.*

Satz 6.10 (μ-rekursive Funktionen = WHILE**-berechenbar)**
Die Menge der μ-rekursive Funktionen entspricht genau der Menge der WHILE*-berechenbaren Funktionen.*

Beweisidee: Wir deuten hier nur kurz an, wie man eine Berechnung einer Turing-Maschine durch eine μ-rekursive Funktion ausdrückt. Es gibt eine primitiv rekursive Funktion f, so daß $f(x,k)$ die (geeignet kodierte) Konfiguration der Turing-Maschine bei Eingabe x nach k Berechnungsschritten liefert. Sei nun g eine Funktion, so daß $g(y)$ für eine kodierte Konfiguration y der Turing-Maschine das Ergebnis 0 liefert, wenn die Turing-Maschine in einem Endzustand ist, und 1 sonst. Dann liefert $\mu(gf)(x)$ die Anzahl der Berechnungsschritte, die die Turing-Maschine bis zur Terminierung auf Eingabe x benötigt. Dann liefert $h(f(x,\mu(gf)(x))$ das Ergebnis der Turing-Maschine, wenn h eine Funktion ist, die aus der kodierten Endkonfiguration das eigentliche Ergebnis der Turing-Maschine berechnet.
Diese Beweisidee zeigt auch, daß man auch hier wieder genau eine μ-Operation braucht, um die Berechnung einer Turing-Maschine zu simulieren. Dies ist das Analogon (bzw. die ursprüngliche Formulierung) der Kleene'schen Normalform für μ-rekursive Funktionen: Jede μ-rekursive Funktion kann mit genau einer Anwendung des μ-Operators definiert werden.

5 Überblick

Nachfolgend sind nochmals die verschiedenen (bzw. gleichen) Berechnungsbegriffe dargestellt, die wir in diesem Kapitel betrachtet haben. Auf der linken Seite sind die korrespondierenden Konzepte aus den Formalen Sprachen angegeben:

Analogie bei Sprachen	**Turing-Maschine**	**Register-Maschine**	**Rekursionen**	**Terminierung**
aufzählbare Sprachen	partiell rekursive Funktionen	Register-Maschinen-, GOTO-, WHILE-berechenbare Funktionen	μ-rekursive Funktionen	Nicht-terminierung möglich
$\cup\!\!\!\!\neq$	$\cup\!\!\!\!\neq$	$\cup\!\!\!\!\neq$	$\cup\!\!\!\!\neq$	
entscheidbare Sprachen	total rekursive Funktionen			Terminierung semantisch gefordert
	$\supset\!\!\!\!\neq$	$\cup\!\!\!\!\neq$	$\cup\!\!\!\!\neq$	
$\cup\!\!\!\!\neq$		FOR-berechenbare Funktionen	primitiv rekursive Funktionen	Terminierung syntaktisch erzwungen
kontextsensitive Sprachen	- Die total rekursiven Funktionen lassen sich nicht syntaktisch Charakterisieren (siehe Übung 13, Aufgabe 2)!			
$\cup\!\!\!\!\neq$				
kontextfreie Sprachen	- Church'sche These: "Mehr als partiell rekursiv geht nicht!"			

Um etwas mehr über Berechenbarkeit zu lernen, sollten Sie mal einen Blick in das Buch von E. Smith [8] werfen. Insbesondere sind dort die hier nur angedeuteten Beweise ausgeführt.

Redaktioneller Hinweis: Das ganze Kapitel 6 ist noch sehr knapp gehalten. Wenigstens die wichtigsten Beweise sollten irgendwann ergänzt werden.

Kapitel 7

Kritik klassischer Berechnungsmodelle

In den beiden vorangegangenen Kapiteln haben wir uns mit den Begriffen des Berechenbaren und des Entscheidbaren beschäftigt. Insbesondere haben wir die Grenzen des Berechenbaren und Entscheidbaren ausgelotet. Wir haben uns also mit dem beschäftigt, was man prinzipiell mit Rechnern machen kann (und was nicht).
Allerdings betrachtet diese „klassiche Berechenbarkeitstheorie“ nur einen kleinen Teil dessen, wofür wir Rechner heute einsetzen: das Berechnen von Funktionen[1]. Viele (möglicherweise die meisten) Dinge, für die wir Rechner heute einsetzen, kann man nur mit Mühe oder gar nicht als die Berechnung einer Funktion auffassen. Beispielsweise lassen sich die folgenden Systeme nur schwer als die Berechnung einer Funktionen auffassen?

- Betriebssystem
- Betriebliches Informationssystem
- Web-Server
- Workflow-Managementsystem

In der theoretischen Informatik könnte man manchmal den Eindruck gewinnen, dass die Welt nur aus dem Berechnen von Funktionen besteht. Die informationstechnische Realität sieht aber oft anders aus. Sie beschäftigt sich mit *interaktiven* oder *reaktiven Systemen*. Diese Systeme unterscheiden sich in zwei wesentlichen Punkten von der Berechnung einer Funktion:

- Die Eingabe liegt nicht bereits am Anfang der „Berechnung“ vor; die „Berechnung“ besteht aus einer Interaktion von Ein- und Ausgaben.
- Für das Systemverhalten ist nicht nur die Beziehung zwischen Eingabe- und Ausgabe relevant, sondern auch das was zwischendurch passiert.
- Die „Berechnung“ muß nicht unbedingt terminieren. Bei vielen Systemen ist die Nicht-Terminierung sogar der gewünschte Fall (wenigstens konzeptuell). Beispielsweise sollte ein Betriebssystem im Idealfall nie terminieren.

[1] Wir haben uns hier sogar auf Funktionen über den natürlichen Zahlen $f : \mathbb{N}^n \to \mathbb{N}$ beschränkt. Das ist aber keine prinzipielle Einschränkung, da man andere Datentypen durch geeignete Kodierung repräsentieren kann. Datentypen sind also – was die prinzipiellen Aspekte der Berechenbarkeit betrifft – „nur“ eine Frage der Kodierung. Aber natürlich gibt es eine eigenständige Theorie der abstrakten Datentypen.

Es gibt zwar Versuche, auch interaktive Systeme als das Berechnen von Funktionen aufzufassen. Aber das ist nicht unbedingt zweckmäßig.
Warum also haben wir uns mit der „klassichen Berechenbarkeitstheorie“ beschäftigt, obwohl die informationstechnische Wirklichkeit ganz anders aussieht? Darauf gibt es verschiedene Antworten:

- Die klassiche Berechenbarkeitstheorie zeigt die prinzipiellen Grenzen des Machbaren auf. Dieser Grenzen sollte sich jeder Informatiker bewußt sein.

- Interaktive Systeme bestehen in vielen Fällen aus Teilkomponenten, die Funktionen berechnen.

- Es gibt Automatenmodelle oder auf Automatenmodellen basierende Formalismen, zur Beschreibung reaktiver Systeme:

  · Petrinetze
  · Prozessalgebren
  · Statecharts
  · Büchi-Automaten
  · I/O-Automaten
  · ...

  } Dazu gibt es Spezialvorlesungen

  Viele der Techniken, die wir in der Vorlesung kennengelernt haben, spielen auch bei diesen erweiterten Automatenmodellen eine Rolle.

Die klassische Berechenbarkeitstheorie hat viel zum Verständnis des mit Rechnern Machbaren beigetragen. Sie nimmt deshalb nach wie vor einen wichtigen Platz in der Informatik ein. Wir sollten uns durch sie aber nicht den Blick für die “informatischen Wirklichkeit” verstellen lassen!

Literaturverzeichnis

Obwohl „Automaten, Formale Sprachen und Berechenbarkeit“ ein klassisches und – im Maßstab der Informatik gemessen – sehr altes Gebiet ist, gibt es bis heute nicht die eine allgemein akzeptierte Darstellung dieses Themas. Die Notationen und auch der Aufbau der Vorlesungen variieren – je nach Schwerpunkt der Vorlesung und nach Vorlieben und Geschmack des Dozenten (je nach dem, ob er eher aus dem Gebiet der Formalen Sprachen, der Automatentheorie, des Compilerbaus, der Komplexitätstheorie oder der Berechenbarkeitstheorie kommt).

Es ist sehr empfehlenswert, sich außer diesem Skript einige weitere Lehrbücher zu diesem Thema anzusehen. Denn erst die Auseinandersetzung mit anderen Darstellungen desselben Sachverhalts und Formalisierungen in anderen Notationen ermöglicht die Trennung des Wesentlichen vom Unwesentlichen dieser Konzepte.

Deshalb folgt hier eine (unvollständige) Liste von Lehrbüchern:

[1] Alfred V, Aho, Ravi Sethi, Jeffrey D. Ullman. Compilerbau (Teil 1). Addison-Wesley 1988.

Deutsche Version des „Drachenbuches“, eines Compilerbau-Klassikers.

[2] Wilfried Brauer. Automatentheorie. Teubner 1984.

[3] Katrin Erk, Lutz Priese. Theoretische Informatik – Eine umfassende Einführung. Springer 2000.

[4] Michael A. Harrison. Introduction to Formal Language Theory. Addison-Wesley 1978.

Das wohl ausführlichste Buch zum Thema.

[5] John E. Hopcroft, Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley 1979. 7

Einer der Klassiker (wenn nicht DER *Klassiker) unter den Lehrbücher zum Thema.*

[6] John E. Hopcroft, Jeffrey D. Ullman. Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie. Addison-Wesley 1989.

Schlechte deutsche Übersetzung des Klassikers, die inzwischen wenigstens etwas verbessert wurde.

[7] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley 2001.

Neu konzipierter Nachfolger des Klassikers [5]. Eine deutsche Version gibt es wohl noch nicht.

[8] Einar Smith. Elementare Berechenbarkeitstheorie. Springer 1996. 5

[9] Franz Stetter. Grundbegriffe der Theoretische Informatik. Springer 1988.

Wird wohl nicht mehr verlegt. Schade.

[10] Uwe Schöning. Theoretische Informatik – kurzgefaßt. Spektrum Akademischer Verlag 1992 (4. Aufl. 2001).

[11] Ingo Wegener. Theoretische Informatik – eine algorithmenorientierte Einführung. Teubner 1993 (2. Aufl. 1999).

Index

V

W

X

Z