Skript zur Vorlesung

Algorithmen und Berechnungskomplexität I

Prof. Dr. Heiko Röglin
Institut für Informatik I

universitätbonn

Wintersemester 2010/11

31. Januar 2011

Inhaltsverzeichnis

Bitte senden Sie Hinweise auf Fehler im Skript und Verbesserungsvorschläge an die E-Mail-Adresse `roeglin@cs.uni-bonn.de`.

Kapitel 5

Automatentheorie und formale Sprachen

In diesem Teil der Vorlesung werden wir uns mit *formalen Sprachen* beschäftigen. Eine solche Sprache ist nichts anderes als eine Menge von Zeichenketten über einem gegebenen Alphabet. So bildet zum Beispiel die Menge aller gültigen Java-Programme eine formale Sprache ebenso wie die Menge aller gültigen HTML-Dateien. Für den Entwurf von Programmiersprachen und Compilern ist es unerlässlich eine formale Beschreibung zu entwickeln, welche Zeichenfolgen ein gültiges Programm darstellen.

Diese Beschreibung muss dergestalt sein, dass der Compiler möglichst effizient die *lexikalische Analyse* und die *syntaktische Analyse* durchführen kann. Die lexikalische Analyse ist der erste Schritt, den ein Compiler durchführt; dabei wird der Quelltext in logisch zusammenhängende *Tokens* wie zum Beispiel Schlüsselwörter, Zahlen und Operatoren zerlegt. Der zweite Schritt ist die syntaktische Analyse, in der überprüft wird, ob der Quelltext ein syntaktisch korrektes Programm ist, und in der der Quelltext in einen sogenannten *Syntaxbaum* umgewandelt wird.

Wir werden zunächst *Grammatiken* kennenlernen. Dabei handelt es sich um Regelsysteme, die beschreiben, wie die Wörter einer Sprache erzeugt werden. Grammatiken werden gemäß ihrer Mächtigkeit in verschiedene Klassen eingeteilt, und wir werden uns mit *regulären* und *kontextfreien Grammatiken* im Detail beschäftigen. Während reguläre Grammatiken bei der lexikalischen Analyse eine große Rolle spielen, kommen kontextfreie Grammatiken bei der Syntaxanalyse zum Einsatz, da alle gängigen Programmiersprachen durch kontextfreie Grammatiken beschrieben werden, wenn man von einigen nicht ganz so wichtigen Details, auf die wir hier nicht eingehen werden, absieht.

5.1 Sprachen, Grammatiken und die Chomsky-Hierarchie

In diesem Kapitel gehen wir stets davon aus, dass Σ eine beliebige endliche Menge ist, die wir auch das *Alphabet* nennen werden. Ein *Wort* ist eine endliche Folge von Zeichen aus dem Alphabet Σ, und mit Σ^* bezeichnen wir die Menge aller Wörter, das heißt die Menge aller endlichen Zeichenfolgen über Σ. Das *leere Wort* ε ist die Zeichenkette der Länge 0, und wir definieren, dass ε zu Σ^* gehört. Wir definieren außerdem $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ als die Menge aller Wörter mit mindestens einem Buchstaben. Nun können wir das zentrale Konzept dieses Kapitels definieren.

Definition 5.1.1. *Eine Menge $L \subseteq \Sigma^*$ bezeichnen wir als* (formale) Sprache über Σ.

Wir benötigen noch die folgenden Schreibweisen.

Definition 5.1.2. *Sei Σ ein Alphabet und sei $a \in \Sigma$ und $n \in \mathbb{N} = \{0, 1, 2, 3, \ldots\}$. Mit a^n bezeichnen wir das Wort $a \cdots a$, in dem der Buchstabe a genau n mal vorkommt. Außerdem bezeichne a^0 das leere Wort ε. Sei $w \in \Sigma^*$ ein Wort, dann bezeichne $|w|$ die Länge von w und $|w|_a$ wie oft der Buchstabe a im Wort w enthalten ist. Außerdem bezeichne w^R das gespiegelte Wort, das heißt für $w = w_1 \ldots w_n$ ist $w^R = w_n \ldots w_1$.*

Fangen wir mit dem sehr einfachen Beispiel $\Sigma = \{1\}$ an, bei dem das Alphabet aus nur einem einzigen Element besteht. Für dieses Beispiel gilt $\Sigma^* = \{\varepsilon, 1, 11, 111, 1111, \ldots\} = \{1^0, 1^1, 1^2, 1^3, \ldots\}$. Sprachen über diesem Alphabet sind zum Beispiel $L = \{1^n \mid n \text{ ist gerade}\}$ und $L = \{1^n \mid n \text{ ist eine Primzahl}\}$.

Ein bei Informatikern sehr beliebtes Alphabet ist $\Sigma = \{0, 1\}$. Über diesem Alphabet kann man beispielsweise die folgenden Sprachen definieren.

- $L = \{01, 1111, 010, \varepsilon\}$: Eine Sprache, die vier willkürlich gewählte Wörter enthält.
- $L = \{0^m 1^n \mid m, n \in \mathbb{N}\}$: Die Sprache aller Wörter, die aus einem Block von Nullen bestehen, der von einem Block Einsen gefolgt wird.
- $L = \{0^n 1^n \mid n \in \mathbb{N}\}$: Die Sprache aller Wörter, die aus einem Block von Nullen bestehen, der von einem Block Einsen gefolgt wird, wobei beide Blöcke die gleiche Länge haben.
- $L = \{w \in \Sigma^* \mid |w|_0 = |w|_1\}$: Die Sprache aller Wörter, die genauso viele Nullen wie Einsen enthalten.
- $L = \{w \in \Sigma^* \mid w = w^R\}$: Die Sprache aller *Palindrome*.

Eine geeignete Alphabetwahl für Programmiersprachen ist es, als Alphabet Σ die Menge aller Unicode-Zeichen zu wählen. Dann kann man L zum Beispiel als die Menge aller gültigen Java-Programme definieren.

Um Sprachen kompakt zu beschreiben, werden wir Grammatiken verwenden.

Definition 5.1.3. *Eine* Grammatik *G besteht aus vier Komponenten (Σ, V, S, P) mit den folgenden Bedeutungen:*

- *Σ ist das endliche Alphabet, über dem die Sprache definiert ist. Wir nennen die Elemente aus Σ auch* Terminalsymbole.
- *V ist eine endliche Menge von* Nichterminalsymbolen, *die disjunkt zu Σ ist.*
- *$S \in V$ ist das* Startsymbol.
- *P ist eine endliche Menge von* Ableitungsregeln. *Dabei ist eine Ableitungsregel ein Paar (l, r) mit $l \in V^+$ und $r \in (V \cup \Sigma)^*$. Wir werden statt (l, r) auch $l \to r$ schreiben. Enthält P zwei Regeln (l, r) und (l, r'), so schreiben wir auch $l \to r, r'$.*

Wörter werden nun wie folgt erzeugt: Starte mit dem Startsymbol S und wende solange Ableitungsregeln an, bis nur noch Terminale vorhanden sind. Das Anwenden einer Ableitungsregel (l, r) bedeutet, dass wir ein Vorkommen von l durch r ersetzen.

Definition 5.1.4. *Für eine Grammatik $G = (\Sigma, V, S, P)$ bezeichnen wir mit $L(G) \subseteq \Sigma^*$ die von ihr erzeugte Sprache, das heißt die Menge aller Wörter, die aus dem Startsymbol mit Hilfe der Ableitungsregeln aus P erzeugt werden können.*

Schauen wir uns Beispiele für Grammatiken an, um diese Definition zu verdeutlichen. Dabei sei stets $\Sigma = \{0, 1\}$ angenommen. Formale Beweise, dass die Grammatiken die behaupteten Sprachen erzeugen, überlassen wir dem Leser.

- Es sei $V = \{S\}$ und P enthalte die Regeln $S \to \varepsilon$ und $S \to 11S$. Für diese Grammatik G gilt $L(G) = \{1^n \mid n \text{ ist gerade}\}$. Eine Ableitung des Wortes 1111 sieht zum Beispiel wie folgt aus:
$$S \to 11S \to 1111S \to 1111\varepsilon = 1111.$$

- Es sei $V = \{S, A, B\}$ und P enthalte die Regeln
$$S \to AB \qquad A \to \varepsilon, 0A \qquad B \to \varepsilon, 1B.$$
Für diese Grammatik G gilt $L(G) = \{0^m 1^n \mid m, n \in \mathbb{N}\}$. Eine Ableitung des Wortes 011 sieht zum Beispiel wie folgt aus:
$$S \to AB \to 0AB \to 0\varepsilon B = 0B \to 01B \to 011B \to 011\varepsilon = 011.$$

- Es sei $V = \{S\}$ und P enthalte die Regeln
$$S \to \varepsilon, 0S1.$$
Für diese Grammatik G gilt $L(G) = \{0^n 1^n \mid n \in \mathbb{N}\}$. Eine Ableitung des Wortes 0011 sieht zum Beispiel wie folgt aus:
$$S \to 0S1 \to 00S11 \to 00\varepsilon 11 = 0011.$$

- Es sei $V = \{S\}$ und P enthalte die Regeln
$$S \to \varepsilon, 0, 1, 0S0, 1S1.$$
Für diese Grammatik G gilt $L(G) = \{w \in \Sigma^* \mid w = w^R\}$. Wir geben beispielhaft eine Ableitung des Palindroms 1101011 an:
$$S \to 1S1 \to 11S11 \to 110S011 \to 1101011.$$

Für eine gegebene Grammatik und ein gegebenes Wort ist es ein wichtiges Problem, zu entscheiden, ob das Wort zu der von der Grammatik erzeugten Sprache gehört, ob also zum Beispiel ein gegebener Quelltext ein syntaktisch korrektes Programm einer bestimmten Programmiersprache darstellt. Dieses Problem wird auch *Wortproblem* genannt und es kann für allgemeine Grammatiken so komplex sein, dass es keine effizienten Algorithmen gibt, um es zu lösen. Deshalb müssen wir die erlaubten Ableitungsregeln einschränken, um eine Klasse von Grammatiken zu erhalten, für die das Wortproblem effizient gelöst werden kann. Auf der anderen Seite müssen wir natürlich aufpassen, dass wir die Grammatiken nicht zu stark einschränken, weil wir ansonsten keine mächtigen Programmiersprachen wie C++ oder Java beschreiben können. Die Chomsky-Hierarchie enthält vier Klassen, die verschiedene Kompromisse zwischen Ausdrucksstärke und Komplexität des Wortproblems bilden.

Definition 5.1.5. *Die* Chomsky-Hierarchie *teilt Grammatiken in die folgenden Klassen ein.*

0. *Grammatiken ohne Einschränkungen heißen* Chomsky-0-Grammatiken.
1. *In* Chomsky-1-Grammatiken *oder* kontextsensitiven Grammatiken *haben alle Ableitungsregeln die Form* $\alpha A\beta \to \alpha\gamma\beta$ *oder* $S \to \varepsilon$, *wobei* $\alpha, \beta \in V^*$, $A \in V$ *und* $\gamma \in (\Sigma \cup V)^+$ *gilt. Außerdem darf* S *auf keiner rechten Seite einer Produktionsregel vorkommen.*
2. *In* Chomsky-2-Grammatiken *oder* kontextfreien Grammatiken *haben alle Ableitungsregeln die Form* $A \to v$ *mit* $A \in V$ *und* $v \in (V \cup \Sigma)^*$.
3. *In* Chomsky-3-Grammatiken, rechtslinearen Grammatiken *oder* regulären Grammatiken *haben alle Ableitungsregeln die Form* $A \to v$ *mit* $A \in V$ *und* $v = \varepsilon$ *oder* $v = aB$ *mit* $a \in \Sigma$ *und* $B \in V$.

In dieser Vorlesung sind nur die regulären und die kontextfreien Grammatiken von Interesse. Die anderen beiden Klassen haben wir nur der Vollständigkeit halber aufgeführt. Wir werden Sprachen, die von regulären oder kontextfreien Grammatiken erzeugt werden, im Folgenden auch *reguläre Sprachen* bzw. *kontextfreie Sprachen* nennen.

5.2 Endliche Automaten

In diesem Abschnitt werden wir sehen, dass das Wortproblem für reguläre Grammatiken sehr effizient gelöst werden kann. Dazu führen wir mit *endlichen Automaten* ein sehr einfaches Rechnermodell ein, und wir zeigen, dass die Sprachen, die von diesem einfachen Rechnermodell entschieden werden können, genau die regulären Sprachen sind. Bevor wir den Zusammenhang zu regulären Grammatiken herstellen, beschäftigen wir uns zunächst im Detail mit endlichen Automaten, die auch für sich genommen ein interessantes Rechnermodell darstellen, das beispielsweise als Modell sequentieller Schaltwerke dient.

Definition 5.2.1. *Ein* endlicher Automat *(deterministic finite automaton, DFA)* M *besteht aus fünf Komponenten* $(Q, \Sigma, \delta, q_0, F)$ *mit den folgenden Bedeutungen:*

- Q *ist eine endliche Menge, die* Zustandsmenge.
- Σ *ist eine endliche Menge, das* Eingabealphabet.
- $\delta : Q \times \Sigma \to Q$ *ist die* Zustandsüberführungsfunktion.
- $q_0 \in Q$ *ist der* Startzustand.
- $F \subseteq Q$ *ist die Menge der* akzeptierenden Zustände.

Ein solcher DFA M erhält als Eingabe ein Wort $w = w_1 \dots w_n$ über dem Alphabet Σ und arbeitet die Buchstaben von links nach rechts ab. Er startet im Startzustand q_0 und wechselt nach dem Lesen des ersten Buchstabens in den Zustand $q_1 := \delta(q_0, w_1)$. Nach dem Lesen des zweiten Buchstabens wechselt der DFA in den Zustand $q_2 := \delta(q_1, w_2)$ und so weiter. Nach dem Lesen des letzten Buchstabens stoppt der DFA im Zustand $q_n := \delta(q_{n-1}, w_n)$. Formal erweitern wir die Funktion δ zu einer Funktion $\delta^* : Q \times \Sigma^* \to Q$, sodass für alle $q \in Q$ und für alle $a \in \Sigma$ gilt

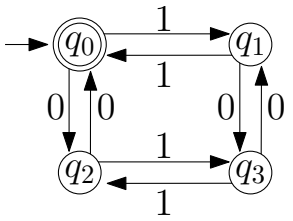
$$\delta^*(q, \varepsilon) = q \quad \text{und} \quad \delta^*(q, a) = \delta(q, a).$$

Außerdem definieren wir rekursiv für jedes Wort $w = w_1 \dots w_n \in \Sigma^*$ der Länge $n \geq 2$

$$\delta^*(q, w) = \delta^*(\delta(q, w_1), w_2 \dots w_n).$$

Erhält der DFA nun ein Wort $w \in \Sigma^*$ als Eingabe, so terminiert er im Zustand $q_n = \delta^*(q_0, w)$. Falls $q_n \in F$ gilt, sagen wir, dass der DFA, das Wort w *akzeptiert*. Gilt $q_n \notin F$, so sagen wir, dass der DFA das Wort w *verwirft*. Mit $L(M) \subseteq \Sigma^*$ bezeichnen wir für einen DFA M die Menge aller Wörter, die M akzeptiert. Wir sagen, dass M die Sprache $L(M)$ *entscheidet* oder *akzeptiert*. Wir sagen auch, dass M ein DFA für die Sprache $L(M)$ ist.

Anschaulich kann man einen DFA anhand eines Übergangsgraphen darstellen. Das folgende Beispiel, in dem $\Sigma = \{0, 1\}$ gilt, ist aus dem Buch von Ingo Wegener übernommen [2]:



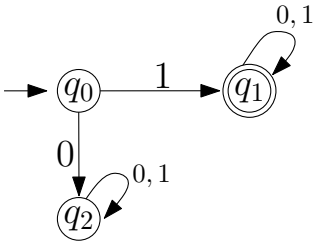
Jeder Zustand aus Q ist durch einen Kreis dargestellt, Zustände aus F sind durch doppelte Kreise dargestellt und der Startzustand ist durch einen eingehenden Pfeil markiert, der von keinem anderen Zustand kommt. Die Funktion δ wird durch die Pfeile zwischen den Zuständen dargestellt. Liest der DFA beispielsweise im Zustand q_0 eine 1, so geht er in den Zustand q_1 über. Liest er im Zustand q_0 eine 0, so geht er in den Zustand q_2 über.

Nun sollten wir noch klären, welche Sprache der Automat M in diesem Beispiel entscheidet. Dazu ist es zunächst wichtig, festzuhalten, dass der einzige akzeptierende Zustand der Zustand q_0 ist. Das heißt, nur wenn der DFA nach dem Lesen aller Zeichen des Eingabewortes w wieder im Startzustand ist, so akzeptiert er w. Man kann nun weiterhin beobachten, dass der Automat sich genau dann in einem der beiden linken Zustände (q_0 oder q_2) befindet, wenn er eine gerade Anzahl an Einsen gelesen hat. Weiterhin befindet sich der Automat genau dann in einem der beiden oberen Zustände (q_0 oder q_1), wenn er eine gerade Anzahl an Nullen gelesen hat. Der DFA befindet sich also genau dann im Zustand q_0, wenn er eine gerade Anzahl an Nullen und eine gerade Anzahl an Einsen gelesen hat. Es gilt also

$$L(M) = \{w \in \Sigma^* \mid |w|_0 \text{ ist gerade und } |w|_1 \text{ ist gerade}\}.$$

Einen formalen Beweis dieser Behauptung überlassen wir auch hier dem Leser.

Es kann auch vorkommen, dass ein DFA beim Lesen eines Zeichens gar keinen Zustandswechsel durchführt. In der grafischen Darstellung bedeutet das, dass es Schleifen gibt, die von einem Knoten zu sich selbst führen. Der folgende DFA akzeptiert beispielsweise genau diejenigen Wörter, die mit einer Eins beginnen.



Im Folgenden werden wir uns damit beschäftigen, wie man Automaten mit möglichst wenig Zuständen konstruiert, wir werden untersuchen, welche Sprachen von DFAs entschieden werden können, und wir werden mit *nichtdeterministischen Automaten* eine Erweiterung von DFAs kennenlernen.

5.2.1 Minimierung endlicher Automaten

Bei den Algorithmen und Datenstrukturen, die wir bisher in der Vorlesung kennengelernt haben, haben wir stets die Laufzeit analysiert, und unser Ziel war es möglichst effiziente Algorithmen und Datenstrukturen zu entwerfen. Die Laufzeit eines DFA ist linear in der Länge des Eingabewortes, da jeder Buchstabe des Eingabewortes zu genau einem Zustandsübergang führt. Bei Automaten ist es stattdessen interessant, die Anzahl der Zustände zu minimieren. Deshalb werden wir uns nun damit beschäftigen, wie man zu einem gegebenen DFA einen DFA mit möglichst wenig Zuständen konstruieren kann, der die gleiche Sprache entscheidet.

Zunächst besteht die Möglichkeit, dass gewisse Zustände eines DFA nicht vom Startzustand aus erreicht werden können. Solche Zustände nennen wir *überflüssig* und wir können sie mit Hilfe einer Tiefensuche finden.

Theorem 5.2.2. *Die Menge der überflüssigen Zustände eines DFA kann in Zeit $O(|Q| \cdot |\Sigma|)$ berechnet werden.*

Beweis. Wir stellen den DFA als gerichteten Graph dar, wobei die Knotenmenge der Zustandsmenge Q entspricht und es eine Kante von $q \in Q$ zu $q' \in Q$ gibt, falls $q' = \delta(q, a)$ für mindestens einen Buchstaben $a \in \Sigma$ gilt. Für einen Zustand q, der in diesem Graphen nicht vom Startzustand q_0 erreicht werden kann, gibt es kein Wort $w \in \Sigma^*$ mit $\delta^*(q_0, w) = q$. Ein solcher Zustand ist also überflüssig. Auf der anderen Seite gibt es für jeden von q_0 erreichbaren Zustand q ein Wort w mit $\delta^*(q_0, w) = q$.

Somit sind die Zustände, die in dem Graphen nicht von q_0 erreicht werden können, genau die überflüssigen Zustände. Die Menge dieser Zustände kann mit einer Tiefensuche gefunden werden, deren Laufzeit $O(n + m)$ für Graphen mit n Knoten und m Kanten beträgt. Der von uns konstruierte Graph hat $n = |Q|$ und höchstens $m = |Q| \cdot |\Sigma|$ Kanten. Somit hat die Tiefensuche eine Laufzeit von $O(|Q| \cdot |\Sigma|)$. □

Wir gehen nun davon aus, dass wir einen Automaten haben, in dem kein Zustand überflüssig ist. Das alleine bedeutet allerdings noch nicht, dass der Automat minimal ist, wie das folgende Beispiel zeigt.

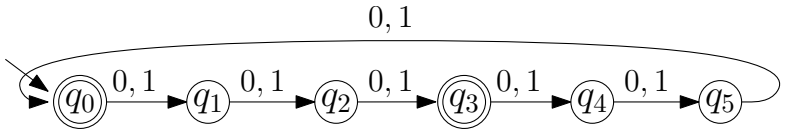


Abbildung 5.1: Beispiel für einen nicht-minimalen DFA ohne überflüssige Zustände

Ähnlich zu dem vorangegangenen Beispiel kann man sich leicht überlegen, dass dieser Automat genau diejenigen Wörter $w \in \{0, 1\}^*$ akzeptiert, die eine Länge haben, die ein Vielfaches

von drei ist. Keiner der Zustände ist überflüssig im Sinne der obigen Definition, aber trotzdem kann der Leser sich leicht selbst davon überzeugen, dass es einen DFA mit nur drei Zuständen gibt, der die gleiche Sprache entscheidet.

Um DFAs zu minimieren, die keine überflüssigen Zustände enthalten, führen wir zunächst die folgende Definition ein.

Definition 5.2.3. *Zwei Zustände p und q eines DFA heißen* äquivalent*, wenn für alle Wörter $w \in \Sigma^*$ gilt:*

$$\delta^*(p, w) \in F \iff \delta^*(q, w) \in F.$$

Wir schreiben dann $p \equiv q$. Mit $[p]$ bezeichnen wir die Menge aller Zustände, die äquivalent zu p sind.

Wenn zwei Zustände p und q äquivalent sind, dann spielt es für das Akzeptanzverhalten des DFA keine Rolle, ob er sich in p oder q befindet. Liest er in p oder q startend das gleiche Wort w, so akzeptiert er entweder in beiden Fällen oder verwirft in beiden Fällen. Man kann sich leicht davon überzeugen, dass $\equiv$ eine Äquivalenzrelation auf der Zustandsmenge Q ist. Das bedeutet es gelten die folgenden drei Eigenschaften, die alle direkt aus der Definition von $\equiv$ folgen:

- Reflexivität: Für alle Zustände $q \in Q$ gilt $q \equiv q$.
- Symmetrie: Gilt $p \equiv q$ für zwei Zustände $p, q \in Q$, so gilt auch $q \equiv p$.
- Transitivität: Gilt $p \equiv q$ und $q \equiv r$ für drei Zustände $p, q, r \in Q$, so gilt auch $p \equiv r$.

Das bedeutet insbesondere, dass die Relation $\equiv$ die Zustandsmenge in Äquivalenzklassen einteilt. Wir konstruieren nun einen DFA, der nur so viele Zustände hat wie die Relation $\equiv$ Äquivalenzklassen besitzt.

Definition 5.2.4. *Die Komponenten des* Äquivalenzklassenautomaten $M' = (Q', \Sigma', \delta', q_0', F')$ *zu einem DFA $M = (Q, \Sigma, \delta, q_0, F)$ sind wie folgt definiert:*

- $Q' = \{[q] \mid q \in Q\}$*: Die Zustände sind genau die Äquivalenzklassen der Relation $\equiv$.*
- $\Sigma' = \Sigma$*: Da der Äquivalenzklassenautomat die gleiche Sprache wie M entscheiden soll, muss er auf dem gleichen Alphabet arbeiten.*
- $q_0' = [q_0]$*: Der Startzustand ist die Äquivalenzklasse des Startzustands.*
- $F' = \{[q] \mid q \in F\}$*: Die akzeptierenden Zustände sind die Äquivalenzklassen der akzeptierenden Zustände von M.*
- $\delta' : Q' \times \Sigma \to Q'$ *mit $\delta'([q], a) = [\delta(q, a)]$ für alle $q \in Q$ und $a \in \Sigma$.*

Die Definition von δ' ist insofern heikel, als dass nicht klar ist, dass sie wohldefiniert ist. Für zwei Zustände $p, q \in Q$ mit $[p] = [q]$ und ein $a \in \Sigma$ könnte beispielsweise $[\delta(p, a)] \neq [\delta(q, a)]$ gelten, in welchem Falle die obige Definition keinen Sinn ergeben würde. Wir zeigen im Folgenden, dass dieser Fall nicht eintreten kann.

Theorem 5.2.5. *Der Äquivalenzklassenautomat M' ist wohldefiniert und entscheidet die gleiche Sprache wie M.*

Beweis. Zunächst zeigen wir, dass δ' wohldefiniert ist. Dazu genügt es zu zeigen, dass für $p, q \in Q$ mit $p \equiv q$ und $a \in \Sigma$ stets $\delta(p,a) \equiv \delta(q,a)$ gilt. Dies folgt aus der Definition von $\equiv$:

$$\begin{aligned} p \equiv q &\Rightarrow \forall w \in \Sigma^* : && [\delta^*(p,w) \in F \iff \delta^*(q,w) \in F] \\ &\Rightarrow \forall a \in \Sigma : \forall w \in \Sigma^* : && [\delta^*(p,aw) \in F \iff \delta^*(q,aw) \in F] \\ &\Rightarrow \forall a \in \Sigma : \forall w \in \Sigma^* : && [\delta^*(\delta(p,a),w) \in F \iff \delta^*(\delta(q,a),w) \in F] \\ &\Rightarrow \forall a \in \Sigma : && \delta(p,a) \equiv \delta(q,a). \end{aligned}$$

Es bleibt noch zu zeigen, dass M' die gleiche Sprache wie M akzeptiert. Als erstes beobachten wir, dass auch die Menge F' in folgendem Sinne wohldefiniert ist: Gilt $p \equiv q$ für zwei Zustände $p, q \in Q$, so sind entweder p und q akzeptierende Zustände oder p und q sind beide keine akzeptierenden Zustände. Dies folgt direkt aus der Definition von $\equiv$, denn $p \equiv q$ impliziert, dass für alle $w \in \Sigma^*$ die Äquivalenz $\delta^*(p,w) \in F \iff \delta^*(q,w) \in F$ gilt. Wir können insbesondere $w = \varepsilon$ einsetzen und erhalten

$$p = \delta^*(p,\varepsilon) \in F \iff q = \delta^*(q,\varepsilon) \in F.$$

Sei nun $w \in \Sigma^*$ und sei $q_0, q_1, \ldots, q_n$ die Zustandsfolge, die der DFA M beim Lesen von w durchläuft. Gemäß der Definition des Äquivalenzklassenautomaten durchläuft M' die Zustandsfolge $[q_0], [q_1], \ldots, [q_n]$. Der DFA M akzeptiert w genau dann, wenn $q_n \in F$ gilt. Der DFA M' akzeptiert w genau dann, wenn $[q_n] \in F'$ gilt. Nach Definition von F' ist aber $[q_n] \in F'$ genau dann, wenn $q_n \in F$. Somit akzeptieren entweder M und M' das Wort w oder beide DFA verwerfen w. □

Der entscheidende Schritt bei der Konstruktion des Äquivalenzklassenautomat M' ist die Berechnung der Äquivalenzklassen der Relation $\equiv$. Wir werden nun einen Algorithmus angeben, der für einen Automaten M diese Äquivalenzklassen berechnet. Sind zwei Zustände p und q nicht äquivalent, so gibt es ein Wort $w \in \Sigma^*$ für das entweder ($\delta^*(p,w) \in F$ und $\delta^*(q,w) \notin F$) oder ($\delta^*(p,w) \notin F$ und $\delta^*(q,w) \in F$) gilt. Ein solches Wort w nennen wir einen *Zeugen* für die Nichtäquivalenz von p und q.

Zunächst ist es nicht ausgeschlossen, dass es nur sehr lange Wörter gibt, die die Nichtäquivalenz zweier Zustände p und q bezeugen. Sei $w \in \Sigma^+$ ein kürzester Zeuge für die Nichtäquivalenz von p und q. Ferner sei $w = aw'$ für $a \in \Sigma$ und $w' \in \Sigma^*$. Dann ist w' ein Zeuge für die Nichtäquivalenz von $\delta(p,a)$ und $\delta(q,a)$. Es muss sich dabei sogar um einen kürzesten solchen Zeugen handeln, denn wäre w'' ein kürzerer Zeuge für die Nichtäquivalenz von $\delta(p,a)$ und $\delta(q,a)$, so wäre aw'' ein kürzerer Zeuge als w für die Nichtäquivalenz von p und q. Diese Beobachtung ist grundlegend für den folgenden Algorithmus und sie impliziert, dass kürzeste Zeugen für die Nichtäquivalenz zweier Zustände nicht zu lang sein können.

Der folgende Algorithmus markiert im Laufe seiner Ausführung Zustandspaare $\{p, q\}$. Wir zeigen unten, dass ein Zustandspaar am Ende des Algorithmus genau dann markiert ist, wenn p und q nicht äquivalent sind. Aus den Markierungen können somit die Äquivalenzklassen der Relation $\equiv$ direkt abgelesen werden. Der Algorithmus wird außerdem für jedes Paar $\{p, q\}$ eine Liste $L(\{p, q\})$ von Zustandspaaren verwalten. Dabei werden nur solche Paare $\{p', q'\}$ in die Liste $L(\{p, q\})$ eingetragen, für die gilt: Wenn p und q nicht äquivalent sind, so sind auch p' und q' nicht äquivalent.

Wir interpretieren diese Listen als Adjazenzlisten eines gerichteten Graphen G, dessen Knotenmenge die Menge aller Paare $\{p, q\} \subseteq Q$ mit $p \neq q$ ist. Eine Kante von $\{p, q\}$ zu $\{p', q'\}$

bedeutet dann, dass p' und q' nicht äquivalent sind, wenn p und q es nicht sind. Wenn in diesem Graphen ein Knoten $\{p, q\}$ markiert ist, so können wir gemäß dieser Interpretation auch alle Knoten $\{p', q'\}$ markieren, die von $\{p, q\}$ aus erreichbar sind.

```
    ERZEUGE-ÄQUIVALENZKLASSENAUTOMAT(M = (Q, Σ, δ, q_0, F))
    // Markiere die Paare {p, q}, für die das leere Wort ihre Nichtäquivalenz bezeugt.
 1  A = ∅;
 2  for each {p, q} {
 3      L({p, q}) = ∅;
 4      if (p ∈ F and q ∉ F) or (p ∉ F and q ∈ F) {
 5          Markiere {p, q};
 6          A.add({p, q});
 7      }
 8  }
    // Erzeuge die Adjazenzlistendarstellung des Graphen G.
 9  for each {p, q} {
10      for each a ∈ Σ {
11          if δ(p, a) ≠ δ(q, a) then L({δ(p, a), δ(q, a)}).add({p, q}));
12      }
13  }
    // Erzeuge die restlichen Markierungen.
14  Markiere mittels Tiefensuche jeden Knoten {p, q} in G, der von einem Knoten in A
    erreichbar ist.
```

Theorem 5.2.6. *Der Algorithmus* ERZEUGE-ÄQUIVALENZKLASSENAUTOMAT *markiert in Zeit* $O(|Q|^2 \cdot |\Sigma|)$ *alle Zustandspaare* $\{p, q\}$ *mit* $p \not\equiv q$.

Beweis. Zunächst überzeugen wir uns davon, dass ein Paar $\{p, q\}$ nur dann einer Liste $L(\{p', q'\})$ hinzugefügt wird, wenn die Nichtäquivalenz von p' und q' die von p und q impliziert. Aus Zeile 11 folgt, dass $p' = \delta(p, a)$ und $q' = \delta(q, a)$ für ein $a \in \Sigma$ gelten muss. Ist nun $w \in \Sigma^*$ ein Zeuge für die Nichtäquivalenz von p' und q', so ist aw ein Zeuge für die Nichtäquivalenz von p und q, da $\delta^*(p', w) = \delta^*(p, aw)$ und $\delta^*(q', w) = \delta^*(q, aw)$ gilt.

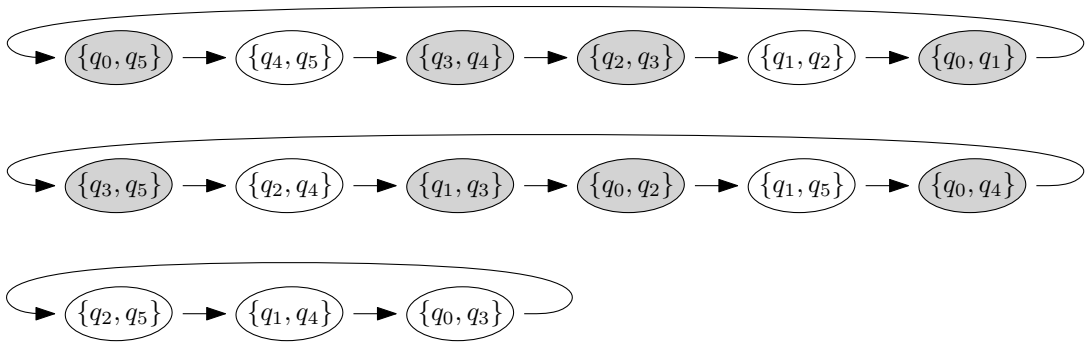
Dies zeigt, dass unsere Interpretation der Kanten im Graphen G korrekt ist. Weiterhin werden in der ersten Phase des Algorithmus in den Zeilen 1 bis 8 nur Zustandspaare markiert, die nicht äquivalent sind; nämlich all jene, für die bereits das leere Wort die Nichtäquivalenz bezeugt. Zusammen bedeutet das, dass der Algorithmus auch in der dritten Phase in Zeile 14 nur Zustandspaare markiert, die nicht äquivalent sind.

Es bleibt zu zeigen, dass alle nicht äquivalenten Zustandspaare markiert werden. Nehmen wir an, es gäbe am Ende des Algorithmus nicht äquivalente Zustandspaare, die nicht markiert sind, und sei $\{p, q\}$ ein solches nicht äquivalentes Zustandspaar, das unter allen nicht markierten nicht äquivalenten Zustandspaaren den kürzesten Zeugen w hat. Wegen der ersten Phase des Algorithmus kann w nicht das leere Wort sein und somit können wir w als $w = aw'$ für ein $a \in \Sigma$ und $w' \in \Sigma^*$ schreiben. Das Wort w' ist dann ein Zeuge für die Nichtäquivalenz von $p' = \delta(p, a)$ und $q' = \delta(q, a)$. Da $\{p, q\}$ unter allen nicht markierten nicht äquivalenten Zustandspaaren den kürzesten Zeugen besitzt, muss das Paar $\{p', q'\}$ markiert sein. Außerdem wird $\{p, q\}$ in Zeile 11 in die Liste $L(\{p', q'\})$ eingefügt. Somit ist $\{p, q\}$ von $\{p', q'\}$ aus

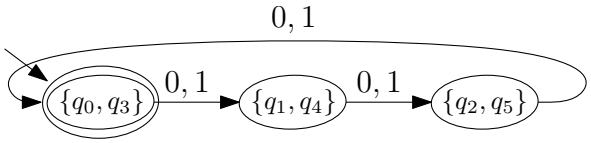
durch eine direkte Kante erreichbar. Wird also $\{p', q'\}$ markiert, so markiert die Tiefensuche in Zeile 14 auch den Knoten $\{p, q\}$ im Widerspruch zu unserer Annahme. Damit ist gezeigt, dass jedes nicht äquivalente Zustandspaar markiert wird und der Algorithmus korrekt ist.

Betrachten wir nun noch die Laufzeit. Für die ersten beiden Phasen brauchen wir lediglich die Anzahl der Iterationen der **for**-Schleifen zu zählen. Für die erste Phase ergibt sich eine Laufzeit von $\Theta(|Q|^2)$ und für die zweite Phase eine Laufzeit $\Theta(|Q|^2 \cdot |\Sigma|)$. Die Laufzeit einer Tiefensuche ist linear in der Größe des Graphen. Der betrachtete Graph hat $\binom{|Q|}{2} = \Theta(|Q|^2)$ Knoten und $O(|Q|^2 \cdot |\Sigma|)$ Kanten. Somit folgt die behauptete Laufzeit. □

Wir wenden den Algorithmus auf den DFA in Abbildung 5.1 an. Nach den ersten beiden Phasen hat der Algorithmus den folgenden Graph konstruiert, in dem die grauen Knoten markiert sind.



In der letzten Phase werden dann alle Knoten markiert, die von den grauen Knoten aus erreichbar sind. Als einzige Knoten bleiben $\{q_2, q_5\}$, $\{q_1, q_4\}$ und $\{q_0, q_3\}$ unmarkiert. Das bedeutet, der Algorithmus hat drei Paare von äquivalenten Zuständen identifiziert. Es ergibt sich der folgende Äquivalenzklassenautomat.



Wir haben nun einen effizienten Algorithmus kennengelernt, der den Äquivalenzklassenautomat M' zu einem gegebenen Automaten M berechnet. Es ist aber nicht klar, dass der Äquivalenzklassenautomat wirklich der Automat mit den wenigsten Zuständen ist, der die gleiche Sprache wie M akzeptiert. Dass dies tatsächlich der Fall ist, werden wir im Rest dieses Abschnittes beweisen.

Theorem 5.2.7. *Der Äquivalenzklassenautomat M' zu einem gegebenen DFA M ohne überflüssige Zustände ist der minimale DFA, der die gleiche Sprache wie M akzeptiert.*

Um dieses Theorem zu beweisen, müssen wir zunächst eine Hilfsaussage beweisen, für die wir den Begriff einer *rechtsinvarianten Äquivalenzrelation* einführen.

Definition 5.2.8. *Sei $R \subseteq \Sigma^* \times \Sigma^*$ eine Äquivalenzrelation. Wir nennen R* rechtsinvariant*, wenn*

$$x\,R\,y \Rightarrow \forall z \in \Sigma^* : xz\,R\,yz.$$

Die Anzahl der Äquivalenzklassen von R nennen wir den Index *von R und schreiben dafür* $\mathrm{index}(R)$.

Wenn wir R_M für einen DFA M so definieren, dass $x \in \Sigma^*$ und $y \in \Sigma^*$ genau dann in Beziehung stehen, wenn sie zum gleichen Zustand führen, also $x\,R\,y \iff \delta^*(q_0, x) = \delta^*(q_0, y)$, so erhalten wir ein einfaches Beispiel für eine rechtsinvariante Äquivalenzrelation.

Ein weiteres wichtiges Beispiel für eine rechtsinvariante Äquivalenzrelation ist die *Nerode-Relation.*

Definition 5.2.9. *Sei $L \subseteq \Sigma^*$ eine Sprache. Zwei Wörter $x, y \in \Sigma^*$ stehen bezüglich der* Nerode-Relation R_L *genau dann in Beziehung (d. h. $x\,R_L\,y$), wenn gilt:*

$$\forall z \in \Sigma^* : xz \in L \iff yz \in L.$$

Dass es sich bei der Nerode-Relation um eine Äquivalenzrelation handelt, kann leicht überprüft werden. Auch ist es nicht schwer einzusehen, dass sie rechtsinvariant ist, denn es gilt:

$$\begin{aligned} x\,R_L\,y \Rightarrow \forall w \in \Sigma^* : &\quad (xw \in L \text{ und } yw \in L) \text{ oder } (xw \notin L \text{ und } yw \notin L) \\ \Rightarrow \forall z \in \Sigma^* : \forall w \in \Sigma^* : &\quad (xzw \in L \text{ und } yzw \in L) \text{ oder } (xzw \notin L \text{ und } yzw \notin L) \\ \Rightarrow \forall z \in \Sigma^* : &\quad xz\,R_L\,yz. \end{aligned}$$

Nun können wir das wesentliche Lemma für den Beweis von Theorem 5.2.7 formulieren.

Lemma 5.2.10. *Sei $L \subseteq \Sigma^*$ eine Sprache. Die folgenden drei Aussagen sind äquivalent.*

a) Es gibt einen DFA, der L entscheidet.

b) L ist die Vereinigung einiger Äquivalenzklassen einer rechtsinvarianten Äquivalenzrelation R mit endlichem Index.

c) Die Nerode-Relation R_L hat einen endlichen Index.

Beweis. Wir beweisen die Äquivalenz der drei Aussagen durch einen Ringschluss.

a) ⇒ b) Sei L eine Sprache, die von einem DFA M entschieden wird. Wir betrachten nun die rechtsinvariante Äquivalenzrelation R_M, die wir in dem Beispiel nach Definition 5.2.8 angegeben haben. Der Index dieser Äquivalenzrelation ist gleich der Anzahl nicht überflüssiger Zustände von M und somit endlich. Die Sprache L ist die Vereinigung aller Äquivalenzklassen von R_M, die zu akzeptierenden Zuständen gehören.

b) ⇒ c) Sei R die rechtsinvariante Äquivalenzrelation mit endlichem Index. Wir zeigen, dass die Nerode-Relation R_L eine Vergröberung von R ist. Formal bedeutet das, dass für alle $x, y \in \Sigma^*$ mit $x\,R\,y$ auch $x\,R_L\,y$ gilt. Das bedeutet, dass die Äquivalenzklassen von R_L Vereinigungen von Äquivalenzklassen von R sind. Somit ist der Index von R_L höchstens so groß wie der von R und damit endlich.

Seien nun $x, y \in \Sigma^*$ mit $x\,R\,y$ gegeben. Da R rechtsinvariant ist, gilt $xz\,R\,yz$ für alle $z \in \Sigma^*$. Da L die Vereinigung von Äquivalenzklassen von R ist, gilt $xz \in L \iff yz \in L$ für alle $z \in \Sigma^*$. Damit folgt gemäß der Definition der Nerode-Relation, dass $x\,R_L\,y$ gilt.

c) ⇒ a) Wir konstruieren einen DFA $M = (Q, \Sigma, \delta, q_0, F)$, der L entscheidet, wie folgt:

$$\begin{aligned} Q &= \text{Menge der Äquivalenzklassen von } R_L, \\ \delta([x], a) &= [xa] \text{ für alle } x \in \Sigma^* \text{ und } a \in \Sigma, \\ q_0 &= [\varepsilon], \\ F &= \{[x] \mid x \in L\}. \end{aligned}$$

Da R_L rechtsinvariant ist, ist die Übergangsfunktion δ wohldefiniert: Für zwei $x, y \in \Sigma^*$ mit $[x] = [y]$ und $a \in \Sigma$ folgt nämlich aus der Rechtsinvarianz $[xa] = [ya]$. Seien $x, y \in \Sigma^*$ mit $y \in [x]$ und $x \in L$ beliebig. Dann gilt auch $y \in L$, denn $x \, R_L \, y$ impliziert insbesondere $x \in L \iff y \in L$. Liest der DFA nun ein Wort $w = w_1 \dots w_n$, so befindet er sich hinterher im Zustand $[w]$, denn

$$\delta^*(q_0, w) = \delta^*([\varepsilon], w_1 \dots w_n) = \delta^*([w_1], w_2 \dots w_n) = \dots = \delta^*([w_1 \dots w_{n-1}], w_n) = [w].$$

Er akzeptiert das Wort w also genau dann, wenn $[w] \in F$ gilt. Nach obiger Überlegung ist das genau dann der Fall, wenn $w \in L$ gilt. □

Wir haben zwei verschiedene Konzepte eingeführt und benutzt: erstens die Äquivalenz von Zuständen eines DFAs und zweitens die Nerode-Relation, die eine Äquivalenzrelation auf der Menge der Wörter aus Σ^* bildet. Aus dem Beweis von Lemma 5.2.10 kann man die folgende Beobachtung über den Zusammenhang dieser beiden Konzepte ableiten.

Beobachtung 5.2.11. *Sei M der minimale DFA für eine Sprache L. Dann entspricht die Äquivalenzrelation R_M, die wir in dem Beispiel nach Definition 5.2.8 angegeben haben, der Nerode-Relation R_L.*

Diese Beobachtung kann auch dazu genutzt werden, die Äquivalenzklassen der Nerode-Relation zu bestimmen, wenn man den minimalen DFA für eine Sprache kennt. Aus dem Beweis von Lemma 5.2.10 folgt auch Theorem 5.2.7.

Beweis von Theorem 5.2.7. Der Beweis ist in zwei Schritte gegliedert. Zunächst zeigen wir, dass der Automat, der in dem Beweisschritt von c) nach a) in Lemma 5.2.10 konstruiert wird, minimal ist. Dann zeigen wir dass der Äquivalenzklassenautomat genauso viele Zustände enthält, also ebenfalls minimal ist.

Sei M ein beliebiger DFA für eine Sprache L mit Zustandsmenge Q'. Aus dem Beweisschritt von a) nach b) in Lemma 5.2.10 folgt, dass L als Vereinigung einiger Äquivalenzklassen einer Äquivalenzrelation R mit Index höchstens $|Q'|$ dargestellt werden kann. Aus dem Beweisschritt von b) nach c) folgt, dass der Index der Nerode-Relation höchstens so groß wie der von R ist. Die Anzahl der Zustände in der Menge Q des Automaten, der im Beweisschritt von c) nach a) konstruiert wird, ist so groß wie der Index von R_L. Zusammen erhalten wir

$$|Q'| = \text{index}(R_L) \leq \text{index}(R) \leq |Q|.$$

Damit muss der DFA, der im Beweisschritt von c) nach a) konstruiert wird, minimal sein, da er höchstens so viele Zustände enthält wie jeder beliebige DFA für L. Insbesondere folgt, dass der Index der Nerode-Relation R_L mit der Anzahl Zustände eines minimalen DFA für L übereinstimmt.

Sei M' der Äquivalenzklassenautomat eines Automaten $M = (Q, \Sigma, \delta, q_0, F)$ ohne überflüssige Zustände, der die Sprache L entscheidet. Wir argumentieren nun, dass die Anzahl Zustände von M' genau gleich dem Index der Nerode-Relation R_L ist. Da M' ebenfalls keine überflüssigen Zustände enthält, genügt es zu zeigen, dass zwei Wörter $x, y \in \Sigma^*$ mit $x \, R_L \, y$ zum gleichen Zustand führen, dass also $\delta^*(q_0, x) \equiv \delta^*(q_0, y)$ gilt. Dies folgt aus den Definition von R_L und der Relation $\equiv$:

$$\begin{aligned}
x \, R_L \, y &\Rightarrow \forall z \in \Sigma^* : && (xz \in L \iff yz \in L) \\
&\Rightarrow \forall z \in \Sigma^* : && (\delta^*(q_0, xz) \in F \iff \delta^*(q_0, yz) \in F) \\
&\Rightarrow \forall z \in \Sigma^* : && (\delta^*(\delta^*(q_0, x), z) \in F \iff \delta^*(\delta^*(q_0, y), z) \in F) \\
&\Rightarrow \delta^*(q_0, x) \equiv \delta^*(q_0, y).
\end{aligned}$$

Damit ist das Theorem bewiesen. □

5.2.2 Pumping-Lemma für endliche Automaten

Eine wichtige Frage für ein Rechnermodell ist, wie mächtig es ist, das heißt, welche Sprachen es entscheiden kann. In diesem Abschnitt möchten wir herausfinden, welche Sprachen von DFAs entschieden werden können, und welche zu komplex für DFAs sind. Dazu werden wir das sogenannte *Pumping-Lemma* kennenlernen, mit dessen Hilfe wir zeigen können, dass es für manche Sprachen keinen DFA gibt.

Fangen wir mit der Sprache $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ an, die wir bereits am Anfang dieses Kapitels als Beispiel gesehen haben. Der Leser sollte an dieser Stelle selbst versuchen, einen DFA für L zu konstruieren. Er wird dabei auf folgendes Problem stoßen: Solange der DFA Nullen liest, muss er sich die Anzahl der bereits gelesenen Nullen merken, um sie später mit der Anzahl der gelesenen Einsen vergleichen zu können. Ein DFA hat aber per Definition nur eine endliche Zustandsmenge und er kann deshalb die Anzahl Nullen für beliebig lange Wörter nicht korrekt zählen. Bei dieser einfachen Sprache können wir diese Intuition formalisieren und zeigen, dass es keinen DFA gibt, der L entscheidet. Angenommen es gibt einen solchen DFA M mit Zustandsmenge Q. Dann betrachten wir die Menge der Wörter $\{\varepsilon, 0, 0^2, \ldots, 0^{|Q|}\}$. Da es sich hierbei um mehr Wörter als Zustände handelt, muss es zwei Wörter 0^i und 0^j mit $i \neq j$ aus dieser Menge geben, nach deren Lesen M im gleichen Zustand ist, also $\delta^*(q_0, 0^i) = \delta^*(q_0, 0^j)$. Da M die Sprache L entscheidet muss $\delta^*(q_0, 0^i 1^i) \in F$ gelten. Damit gilt auch

$$\delta^*(q_0, 0^j 1^i) = \delta^*(\delta^*(q_0, 0^j), 1^i) = \delta^*(\delta^*(q_0, 0^i), 1^i) = \delta^*(q_0, 0^i 1^i) \in F.$$

Damit akzeptiert M das Wort $0^j 1^i$ für $j \neq i$ und entscheidet somit nicht die Sprache L.

Im Allgemeinen ist ein solcher direkter Beweis jedoch schwierig. Eine andere Möglichkeit, um zu zeigen, dass eine Sprache L von keinem DFA entschieden wird, ist zu zeigen, dass der Index der Nerode-Relation R_L nicht endlich ist. Dann folgt die gewünschte Aussage aus Lemma 5.2.10. Aber auch das zu beweisen ist im Allgemeinen schwierig. Das folgende *Pumping-Lemma* liefert hingegen eine notwendige Bedingung dafür, dass L von einem DFA entschieden werden kann, die für viele Sprachen leicht falsifiziert werden kann. Mit dem Pumping-Lemma kann man also für viele Sprachen einfach zeigen, dass sie von keinem DFA entschieden werden. Es trägt seinen Namen, da es zeigt, dass man bei jedem Wort $w \in L$, das zu einer Sprache gehört, die von einem DFA entschieden wird, gewisse Teile vervielfältigen kann ohne dabei die Sprache L zu verlassen. Man kann also in gewisser Weise die Wörter in L „aufpumpen“.

Lemma 5.2.12 (Pumping-Lemma)**.** *Sei L eine Sprache, die von einem DFA entschieden wird. Dann gibt es eine Konstante n, sodass für alle $z \in L$ mit $|z| \geq n$ gilt: Das Wort z kann in drei Teile $z = uvw$ mit $|uv| \leq n$ und $|v| \geq 1$ zerlegt werden, sodass $uv^i w \in L$ für alle $i \geq 0$ gilt.*

Beweis. Es sei M ein DFA für L mit Zustandsmenge Q und es sei $n = |Q|$. Wenn $z \in \Sigma^*$ die Länge mindestens n hat, so durchläuft der Automat M beim Lesen von z mindestens $n + 1$ Zustände. Somit muss mindestens ein Zustand q doppelt auftreten. Sei u das Teilwort von z, nach dessen Lesen M zum ersten Mal in Zustand q ist, und sei uv das Teilwort von z, nach dessen Lesen M zum zweiten Mal im Zustand q ist. Dann gilt gemäß obiger Argumentation $|uv| \leq n$ und $|v| \geq 1$. Das Wort w sei nun einfach so gewählt, dass $z = uvw$ gilt. Die wichtige

Eigenschaft ist, dass M, wenn er im Zustand q startet und das Wort v liest, zum Zustand q zurückkehrt.

Sei $q' = \delta^*(q_0, z) \in F$ der Zustand, den M nach dem Lesen von uvw erreicht. Wir können diesen Zustand auch als $q' = \delta^*(q, w)$ schreiben. Für die obige Wahl von u, v und w und für alle $i \geq 0$ gilt $uv^iw \in L$, denn

$$\delta^*(q_0, uv^iw) = \delta^*(q, v^iw) = \ldots = \delta^*(q, vw) = \delta^*(q, w) = q' \in F.$$

Stellen wir uns den DFA wieder als Graphen vor, so läuft dieser beim Lesen von u zunächst zum Zustand q. Beim Lesen von v läuft er im Kreis wieder zu q zurück. Wie oft dieser Kreis durchlaufen wird, spielt für den DFA keine Rolle, denn anschließend läuft er beim Lesen von w immer zum gleichen Zustand q'. □

Wir wenden das Pumping-Lemma auf zwei Beispiele an. Zunächst betrachten wir wieder die Sprache $L = \{0^i1^i \mid i \in \mathbb{N}\}$. Angenommen es gibt einen DFA für L. Sei dann n die Konstante aus dem Pumping-Lemma und sei $z := 0^n1^n \in L$. Sei nun $z = uvw$ mit $|uv| \leq n$ und $|v| \geq 1$ eine beliebige Zerlegung dieses Wortes. Wir betrachten nun das Wort uv^2w. Da $|uv| \leq n$ ist und z mit n Nullen beginnt, gilt $v = 0^{|v|}$. Demzufolge gilt $uv^2w = 0^{n+|v|}1^n$. Wegen $|v| \geq 1$ gehört dieses Wort nicht zu L. Somit ist gezeigt, dass es keinen DFA für L gibt.

Nun betrachten wir noch die Sprache $L = \{w \in \{0,1\}^* \mid w = w^R\}$ aller Palindrome. Angenommen es gibt einen DFA für L. Sei dann n die Konstante aus dem Pumping-Lemma und sei $z := 0^n10^n \in L$. Sei nun $z = uvw$ mit $|uv| \leq n$ und $|v| \geq 1$ eine beliebige Zerlegung dieses Wortes. Wir betrachten nun das Wort uv^2w. Da $|uv| \leq n$ ist und z mit n Nullen beginnt, gilt $v = 0^{|v|}$. Demzufolge gilt $uv^2w = 0^{n+|v|}10^n$. Wegen $|v| \geq 1$ gehört dieses Wort nicht zu L. Somit ist gezeigt, dass es keinen DFA für L gibt.

Exkurs: Das Pumping-Lemma als Spiel

Da das Pumping-Lemma von Studierenden, die es zum ersten Mal sehen, oft als schwer empfunden wird, wollen wir es in diesem Exkurs noch einmal in etwas anderer Gestalt vorstellen. Wir können es mit Hilfe von Quantoren wie folgt ausdrücken:

$$\begin{aligned} &L \subseteq \Sigma^* \text{ ist regulär} \Rightarrow \\ &\quad \exists n \in \mathbb{N}: \quad \forall z \in L, |z| \geq n: \quad \exists \text{ Zerlegung } z = uvw, |uv| \leq n, |v| \geq 1: \quad \forall i \geq 0: uv^iw \in L \end{aligned}$$

Diese Aussage wird in der Regel dafür benutzt, zu zeigen, dass eine Sprache von keinem DFA entschieden wird. Man zeigt also, dass die Aussage rechts vom Implikationspfeil nicht gilt (dass also ihre Negation gilt) und folgert daraus, dass L nicht regulär sein kann:

$$\begin{aligned} &\forall n \in \mathbb{N}: \\ &\qquad \exists z \in L, |z| \geq n: \\ &\qquad\qquad \forall \text{ Zerlegung } z = uvw, |uv| \leq n, |v| \geq 1: \\ &\qquad\qquad\qquad \exists i \geq 0: uv^iw \notin L: \\ &\Rightarrow \text{L ist nicht regulär} \end{aligned}$$

Wie zeigt man nun aber, dass diese Negation des Pumping Lemmas gilt? Man kann sich das als ein Spiel zweier Personen vorstellen. Peter (Prover) möchte Vera (Verifier) davon

überzeugen, dass die Aussage gilt. Vera möchte sich aber nur davon überzeugen lassen, wenn die Aussage wirklich wahr ist. Vera übernimmt die ∀-Rolle und Peter die ∃-Rolle. Das Spiel sieht wie folgt aus:

1. Runde: Vera wählt ein $n \in \mathbb{N}$.
2. Runde: Peter wählt ein $z \in L$ mit $|z| \geq n$.
3. Runde: Vera wählt eine Zerlegung $z = uvw$ mit $|uv| \leq n$ und $|v| \geq 1$.
4. Runde: Peter wählt ein $i \geq 0$.

Gelingt es Peter, in der 4. Runde ein i so zu wählen, dass $uv^iw \notin L$ gilt, so hat er gewonnen. Hat er eine Strategie, auf jede Wahl, die Vera in den Runden 1 und 3 trifft, zu reagieren, und schafft es am Ende immer, ein i zu wählen, so dass $uv^iw \notin L$ gilt, so gilt die Negation des Pumping-Lemmas und die Sprache ist nicht regulär.

Wir betrachten wieder die Sprache $L = \{0^i1^i \mid i \in \mathbb{N}\}$ als Beispiel. Wir wollen zeigen, dass L nicht regulär ist und übernehmen Peters Rolle.

1. Runde: Vera wählt ein beliebiges $n \in \mathbb{N}$.
2. Runde: Wir wählen $z = 0^n1^n$. Offenbar gilt $z \in L$ und $|z| \geq n$.
3. Runde: Vera wählt eine beliebige Zerlegung $z = uvw = 0^n1^n$, wobei $|uv| \leq n$ und $|v| \geq 1$ gilt.
4.Runde: Da uv höchstens die Länge n haben darf und das Wort z mit n Einsen beginnt, gilt $uv = 0^j$ für ein $j \leq n$. Auf die gleiche Weise folgt $v = 0^k$ für ein $k \geq 1$. Wir wählen nun $i = 2$.

Betrachten wir das Wort uv^2w. Mit unseren Vorüberlegungen gilt $uv^2w = 0^{j-k}0^{2k}0^{n-j}1^n = 0^{n+k}1^n$. Da $k \geq 1$ gilt, gilt $0^{n+k}1^n \notin L$. Wir haben uns also auf ein beliebiges n und eine beliebige Zerlegung eingestellt und in jedem Fall ein Wort uv^iw gefunden, das nicht zur Sprache gehört. Also haben wir gewonnen und somit ist die Sprache nicht regulär.

5.2.3 Nichtdeterministische endliche Automaten

Ein in der theoretischen Informatik sehr zentraler Begriff ist der des *Nichtdeterminismus*. Wir werden diesen Begriff im zweiten Teil der Vorlesung ausführlich behandeln und deshalb ist es gut, sich bereits jetzt bei dem einfachen Modell der endlichen Automaten mit ihm vertraut zu machen.

Definition 5.2.13. *Ein* nichtdeterministischer endlicher Automat *(nondeterministic finite automaton, NFA) M besteht aus fünf Komponenten $(Q, \Sigma, \delta, q_0, F)$. Der einzige Unterschied zu einem DFA ist die Zustandsüberführungsfunktion, die bei einem NFA eine* Zustandsüberführungsrelation *ist. Es ist also $\delta \subseteq (Q \times \Sigma) \times Q$.*

Wir können δ ebenso als eine Abbildung von $Q \times \Sigma$ in die Potenzmenge von Q auffassen. Dann ist $\delta(q, a)$ die Menge aller Zustände q', für die $((q, a), q') \in \delta$ gilt. Wenn der Automat im Zustand q das Zeichen a liest, so ist der Nachfolgezustand q' nicht mehr eindeutig, sondern es gibt stattdessen eine Menge von möglichen Nachfolgezuständen, die erreicht werden können. Genau wie bei DFAs erweitern wir diese Abbildung von $Q \times \Sigma$ in die Potenzmenge von Q

zu einer Abbildung δ^* von $Q \times \Sigma^*$ in die Potenzmenge von Q. Dabei sei für alle $q \in Q$ und $a \in \Sigma$

$$\delta^*(q, \varepsilon) = \{q\} \quad \text{und} \quad \delta^*(q, a) = \delta(q, a).$$

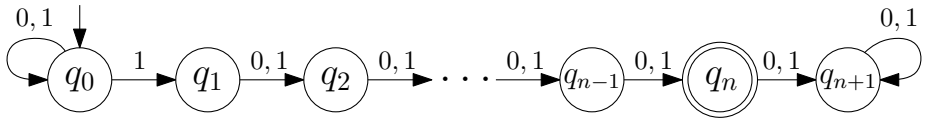
Außerdem definieren wir rekursiv für jedes Wort $w = w_1 \dots w_n \in \Sigma^*$ der Länge $n \geq 2$

$$\delta^*(q, w) = \bigcup_{p \in \delta(q, w_1)} \delta^*(p, w_2 \dots w_n).$$

Erhält der NFA nun ein Wort $w \in \Sigma^*$ als Eingabe, so ist $\delta^*(q_0, w)$ die Menge der Zustände, die er erreichen kann. Falls $\delta^*(q_0, w) \cap F \neq \emptyset$ gilt, falls also die Möglichkeit besteht, dass der NFA nach dem Lesen von w einen akzeptierenden Zustand erreicht, so sagen wir, dass der NFA, das Wort w *akzeptiert*. Ansonsten, so sagen wir, dass der NFA das Wort w *verwirft*. Mit $L(M) \subseteq \Sigma^*$ bezeichnen wir für einen NFA M die Menge aller Wörter, die M akzeptiert. Wir sagen, dass M die Sprache $L(M)$ *entscheidet* oder *akzeptiert*. Wir sagen auch, dass M ein NFA für die Sprache $L(M)$ ist.

Wir werden gleich sehen, dass es Sprachen gibt, für die der minimale NFA sehr viel weniger Zustände besitzt als der minimale DFA. In diesem Sinne können NFAs benutzt werden, um manche Sprachen kompakter zu beschreiben. Wir werden jedoch auch sehen, dass DFAs und NFAs in dem Sinne gleich mächtig sind, dass es zu jedem NFA einen DFA gibt, der die gleiche Sprache entscheidet (wenn auch mit mehr Zuständen). Wir werden NFAs hauptsächlich als theoretisches Hilfsmittel in einigen der folgenden Überlegungen einsetzen.

Auch NFAs können wir uns wieder als Übergangsgraphen vorstellen. Der einzige Unterschied zu DFAs ist, dass nun ein Knoten mehrere ausgehende Kanten haben kann, die mit dem gleichen Zeichen aus Σ beschriftet sind. Betrachten wir das folgende Beispiel:



In diesem Beispiel ist der Zustand q_0 der einzige Zustand, bei dem vom Nichtdeterminismus Gebrauch gemacht wird: Wird in diesem Zustand eine Eins gelesen, so kann sich der NFA entscheiden, ob er im Zustand q_0 bleibt, oder ob er in den Zustand q_1 wechselt. Alle anderen Zustandsübergänge sind deterministisch. Die Sprache, die dieser NFA entscheidet, ist

$$L = \{w \in \{0,1\}^* \mid \text{das } n\text{-letzte Zeichen von } w \text{ ist } 1\}.$$

Um dies zu zeigen, sei zunächst w ein Wort, dessen n-letztes Zeichen 1 ist. Dieses Wort können wir als $w = u1v$ mit $u \in \{0,1\}^*$ und $v \in \{0,1\}^{n-1}$ schreiben. Der NFA kann dann beim Lesen von u im Zustand q_0 bleiben und danach beim Lesen der Eins in den Zustand q_1 wechseln. Dann wechselt er anschließend deterministisch beim Lesen von v in den akzeptierenden Zustand q_n. Somit akzeptiert der NFA das Wort w, da er den Zustand q_n erreichen kann. Nun müssen wir noch zeigen, dass er kein Wort $w \in \{0,1\}^*$ akzeptiert, dessen n-letztes Zeichen eine Null ist. Sei $w = u0v$ mit $u \in \{0,1\}^*$ und $v \in \{0,1\}^{n-1}$. Um nach dem Lesen von w im einzigen akzeptierenden Zustand q_n zu sein, muss der NFA genau beim Lesen des n-letzten Zeichens vom Zustand q_0 in den Zustand q_1 wechseln. Wechselt er früher von q_0 nach q_1, so erreicht er den Zustand q_{n+1}, wechselt er später oder gar nicht, so erreicht er einen Zustand q_i mit $i < n$. Der Wechsel von q_0 nach q_1 beim Lesen des n-letzten Zeichens ist aber nicht möglich, da dies eine Null ist. Somit gibt es keine Möglichkeit für den NFA nach dem Lesen von w in einem akzeptierenden Zustand zu sein.

Bei der Konstruktion eines DFA für L besteht die Schwierigkeit darin, dass wir beim Lesen eines Zeichens noch nicht wissen, ob es das n-letzte Zeichen ist. Beim NFA konnten wir dieses Problem einfach dadurch umgehen, dass wir dem NFA zwei mögliche Zustandsübergänge gegeben haben, wenn er im Startzustand eine Eins liest. Er hat dann gewissermaßen die Möglichkeit, zu raten, ob es sich bei der gerade gelesenen Eins um das n-letzte Zeichen handelt. Rät er richtig, so erreicht er den akzeptierenden Zustand. Tatsächlich können wir zeigen, dass jeder DFA für die Sprache L deutlich mehr Zustände haben muss.

Theorem 5.2.14. *Jeder DFA für die oben definierte Sprache L hat mindestens 2^n Zustände.*

Beweis. Gemäß dem Beweis von Theorem 5.2.7 genügt es zu zeigen, dass die Nerode-Relation R_L von L einen Index von mindestens 2^n hat. Dazu genügt es zu zeigen, dass alle Wörter $x \in \{0,1\}^n$ in unterschiedlichen Äquivalenzklassen der Nerode-Relation liegen. Seien $x, y \in \{0,1\}^n$ mit $x \neq y$ gegeben. Dann gibt es einen Index $i \in \{1, \ldots, n\}$ mit $x_i \neq y_i$. Nehmen wir ohne Beschränkung der Allgemeinheit an, dass $x_i = 0$ und $y_i = 1$ gilt. Es gilt dann $x0^{i-1} \notin L$, weil $x_i = 0$ der n-letzte Buchstabe von $x0^{i-1}$ ist. Andererseits gilt aber aus dem gleichen Grund $y0^{i-1} \in L$. Damit ist gezeigt, dass x und y nicht in Relation bezüglich der Nerode-Relation R_L stehen. □

Es stellt sich nun sofort die Frage, ob es eine Sprache gibt, die von einem NFA entschieden wird, für die es aber keinen DFA gibt. Das folgende Theorem zeigt, dass dies nicht der Fall ist.

Theorem 5.2.15. *Zu jedem NFA mit n Zuständen gibt es einen DFA mit 2^n Zuständen, der die gleiche Sprache entscheidet.*

Beweis. Wir beweisen diesen Satz, indem wir für einen beliebigen NFA $M = (Q, \Sigma, \delta, q_0, F)$ mit $n = |Q|$ einen DFA $M' = (Q', \Sigma, \delta', q_0', F')$ mit $L(M) = L(M')$ und $|Q'| = 2^n$ konstruieren. Für diesen DFA gilt:

- Q' ist die Potenzmenge von Q,
- $q_0' = \{q_0\}$,
- $F' = \{q \subseteq Q' \mid q \cap F \neq \emptyset\}$,
- $\delta' : Q' \times \Sigma \to Q'$ ist für $q \in Q'$ und $a \in \Sigma$ definiert als

$$\delta'(q, a) = \bigcup_{p \in q} \delta(p, a).$$

Diese Konstruktion wird auch *Potenzmengenkonstruktion* genannt, und es gilt wie gewünscht $|Q'| = 2^n$. Wir müssen nur noch zeigen, dass M' die gleiche Sprache wie M entscheidet. Dazu genügt es zu zeigen, dass die Menge $(\delta')^*(q_0, w)$ genau diejenigen Zustände enthält, die der Automat M beim Lesen des Eingabewortes w erreichen kann. Dies folgt unmittelbar aus der Definition von δ'. Formal kann es durch eine Induktion über die Länge von w gezeigt werden. □

Damit haben wir gezeigt, dass jede Sprache, die von einem NFA entschieden werden kann, auch von einem DFA entschieden werden kann. Andersherum sind DFAs nur spezielle NFAs und somit kann auch jede Sprache, die von einem DFA entschieden wird, von einem NFA entschieden werden. Somit sind diese beiden Klassen von Sprachen identisch und DFAs und NFAs sind gleich mächtig.

5.3 Reguläre Sprachen, endliche Automaten und reguläre Ausdrücke

Wir kommen jetzt zum Beginn dieses Kapitels zurück und stellen den Zusammenhang zwischen regulären Sprachen und endlichen Automaten her. Einer der wesentlichen Gründe, warum wir uns mit Automaten beschäftigt haben, ist das folgende Theorem.

Theorem 5.3.1. *Die Klasse der Sprachen, die von DFAs entschieden werden können, ist genau die Klasse der regulären Sprachen.*

Beweis. Zuerst zeigen wir, wie man für einen DFA $M = (Q, \Sigma, \delta, q_0, F)$ eine reguläre Grammatik $G = (\Sigma, V, S, P)$ konstruieren kann, für die $L(G) = L(M)$ gilt. Die Idee besteht darin, mit Hilfe der regulären Grammatik die Berechnung des DFA zu simulieren. Dazu setzen wir $V = Q$ und $S = q_0$. Die Menge P enthält für alle $q, q' \in Q$ und $a \in \Sigma$ mit $\delta(q, a) = q'$ eine Regel $q \to aq'$. Des Weiteren gibt es in P die Regel $q \to \varepsilon$ für alle $q \in F$.

Sei $w = w_1 \dots w_n \in L(M)$ ein Wort, das der DFA M akzeptiert, und sei $q_0, q_1, \dots, q_n$ die Zustandsfolge, die M beim Lesen von w durchläuft. Dann gilt $\delta(q_{i-1}, w_i) = q_i$ für alle $i \geq 1$ und $q_n \in F$. Demzufolge enthält P die Ableitungsregeln $q_{i-1} \to w_i q_i$ sowie $q_n \to \varepsilon$ und wir können das Wort w wie folgt ableiten:

$$q_0 \to w_1 q_1 \to w_1 w_2 q_2 \to \dots \to w_1 \dots w_n q_n \to w_1 \dots w_n \varepsilon = w.$$

Sei umgekehrt $w = w_1 \dots w_n \in L(G)$ ein Wort, das wir mit der Grammatik ableiten können. Wegen der eingeschränkten Regeln der Grammatik muss die Ableitung von w wieder die obige Form für eine Zustandsfolge $q_0, \dots, q_n$ mit $q_n \in F$ haben. Weiterhin muss für diese Zustandsfolge wegen der Definition von P wieder $q_{i-1} \to w_i q_i$ für alle $i \geq 1$ gelten. Demzufolge gilt $\delta(q_{i-1}, w_i) = q_i$ für alle $i \geq 1$ und somit akzeptiert auch der DFA M das Wort w. Insgesamt haben wir gezeigt, dass tatsächlich $L(G) = L(M)$ gilt.

Nun zeigen wir, wie man für eine reguläre Grammatik $G = (\Sigma, V, S, P)$ einen DFA konstruieren kann, der die Sprache $L(G)$ entscheidet. Nach Theorem 5.2.15 genügt es, einen NFA $M = (Q, \Sigma, \delta, q_0, F)$ mit $L(M) = L(G)$ zu konstruieren. Wir setzen $Q = V$, $q_0 = S$ und $F = \{A \in V \mid (A \to \varepsilon) \in P\}$. Außerdem sei

$$\delta(A, a) = \{B \mid (A \to aB) \in P\}.$$

Sei $w = w_1 \dots w_n \in L(G)$. Dann gibt es eine Ableitung

$$S \to w_1 A_1 \to w_1 w_2 A_2 \to \dots \to w_1 \dots w_n A_n \to w_1 \dots w_n \varepsilon = w.$$

Es gibt also die Regeln $S \to w_1 A_1$ und $A_{i-1} \to w_i A_i$ für $i \geq 2$ in P. Des Weiteren gibt es die Regel $A_n \to \varepsilon$ in P. Dementsprechend gilt $A_1 \in \delta(S, w_1)$ und $A_i \in \delta(A_{i-1}, w_i)$ für $i \geq 2$. Erhält der NFA M das Wort w als Eingabe, so kann er also die Zustandsfolge $S, A_1, \dots, A_n$ durchlaufen. Wegen $(A_n \to \varepsilon) \in P$ ist $A_n \in F$. Somit akzeptiert der NFA M das Wort w.

Sei umgekehrt $w = w_1 \dots w_n \in L(M)$ ein Wort, das der NFA M akzeptiert. Dann gibt es eine Folge $S, A_1, \dots, A_n$ von Zuständen mit $A_n \in F$ und mit $A_1 \in \delta(S, w_1)$ und $A_i \in \delta(A_{i-1}, w_i)$ für $i \geq 2$. Dementsprechend muss es in P die Ableitungsregeln $S \to w_1 A_1$, $A_{i-1} \to w_i A_i$ für $i \geq 2$ und $A_n \to \varepsilon$ geben. Damit gilt auch $w \in L(G)$:

$$S \to w_1 A_1 \to w_1 w_2 A_2 \to \dots \to w_1 \dots w_n A_n \to w_1 \dots w_n \varepsilon = w. \qquad \square$$

Es gibt noch eine weitere gängige Möglichkeit, die Klasse der regulären Sprachen zu charakterisieren, nämlich über sogenannte *regulären Ausdrücke*.

Definition 5.3.2. *Sei Σ ein endliches Alphabet. Die Menge der* regulären Ausdrücke *über Σ ist genau die Menge der Ausdrücke, die sich aus den folgenden beiden Regeln erzeugen lassen.*

a) Die Ausdrücke $\emptyset$, ε und a für $a \in \Sigma$ sind reguläre Ausdrücke. Dabei ist $\emptyset$ ein regulärer Ausdruck, der die leere Sprache $\emptyset \subseteq \Sigma^*$ beschreibt, ε ist ein regulärer Ausdruck, der die Sprache $\{\varepsilon\}$ beschreibt, die nur aus dem leeren Wort besteht, und a ist ein regulärer Ausdruck, der die Sprache $\{a\}$ beschreibt, die nur aus dem Wort a besteht.

b) Sind A_1 und A_2 reguläre Ausdrücke, die die Sprachen L_1 und L_2 beschreiben, dann gilt:

- *$(A_1) + (A_2)$ ist ein regulärer Ausdruck für die Sprache $L_1 \cup L_2$.*
- *$(A_1) \cdot (A_2)$ ist ein regulärer Ausdruck für die Sprache $L_1 \cdot L_2 := \{w_1 w_2 \in \Sigma^* \mid w_1 \in L_1, w_2 \in L_2\}$, die sogenannte* Konkatenation *von L_1 und L_2.*
- *$(A_1)^*$ ist ein regulärer Ausdruck für den* Kleeneschen Abschluss *L_1^* von L_1. Dabei handelt es sich um die Sprache $L_1^* = \cup_{i \geq 0} L_1^i$ mit $L_1^0 = \{\varepsilon\}$ und $L_1^i = L_1 \cdot L_1^{i-1}$ für $i \geq 1$.*

Wir sparen Klammern, indem wir die Klammern um die elementaren Ausdrücke $\emptyset$, ε und a weglassen und indem wir definieren, dass * eine höhere Priorität als $\cdot$ hat, was wiederum eine höhere Priorität als $+$ hat. Außerdem lassen wir wie bei der Multiplikation von Zahlen und Variablen oft das Zeichen $\cdot$ weg. Dann ist

$$0(01)^* + (01 + 10)^*$$

zum Beispiel ein regulärer Ausdruck für die Sprache aller Wörter $w \in \{0,1\}^*$, die entweder mit einer Null beginnen und anschließend beliebig viele Wiederholungen von 01 enthalten oder die aus beliebig vielen Wiederholungen der Blöcke 01 und 10 in einer beliebigen Reihenfolge bestehen.

Leser, die mit dem Unix-Programm `grep` vertraut sind, haben reguläre Ausdrücke bereits gesehen. `grep` kann man als Parameter nämlich einen regulären Ausdruck übergeben und es sucht dann in Dateien nach Wörtern, die zu der Sprache dieses regulären Ausdrucks gehören.

Das folgende Theorem werden wir aus Zeitgründen in der Vorlesung nicht beweisen.

Theorem 5.3.3. *Die Klasse der Sprachen, die mit regulären Ausdrücken beschrieben werden können, ist genau die Klasse der regulären Sprachen.*

5.4 Kontextfreie Sprachen

Wir wenden uns nun den kontextfreien Sprachen zu. Diese Sprachen sind besonders deshalb wichtig, weil durch sie alle gängigen Programmiersprachen beschrieben werden können, wenn man von einigen nicht ganz so wichtigen Details absieht. Wir werden uns zunächst mit dem Wortproblem für kontextfreie Sprachen beschäftigen und einen Algorithmus angeben, der für eine gegebene kontextfreie Grammatik G und ein Wort w entscheidet, ob $w \in L(G)$ gilt. Dieser Algorithmus kann also zum Beispiel dazu benutzt werden, um zu entscheiden, ob ein gegebener Quelltext ein syntaktisch korrektes Java-Programm darstellt. Anschließend

werden wir untersuchen, wie mächtig kontextfreie Grammatiken sind und welche Sprachen sie nicht beschreiben können. Dazu werden wir eine Variante des Pumping-Lemma für kontextfreie Sprachen kennenlernen. Zum Schluss werden wir auch für kontextfreie Sprachen ein Automatenmodell einführen, mit dem man genau die kontextfreien Sprachen entscheiden kann. Bei diesem Modell handelt es sich um eine Erweiterung von endlichen Automaten.

Als erste Beobachtung können wir festhalten, dass kontextfreie Grammatiken tatsächlich mächtiger sind als reguläre Grammatiken. Ein einfaches Beispiel dafür ist die Sprache $L = \{0^n 1^n \mid n \in \mathbb{N}\}$, von der wir mit Hilfe des Pumping-Lemmas gezeigt haben, dass sie nicht regulär ist, für die wir aber nach Definition 5.1.4 eine kontextfreie Grammatik angegeben haben.

Um die Ableitung von Wörtern einer kontextfreien Sprache zu visualisieren, bietet sich ein *Syntaxbaum* an. An der Wurzel eines solchen Baumes steht das Startsymbol, jeder innere Knoten ist entweder mit einem Zeichen aus Σ oder einer Variablen aus V beschriftet und jedes Blatt ist mit einem Zeichen aus Σ oder mit ε beschriftet. Ist ein innerer Knoten mit $A \in V$ beschriftet und sind seine Söhne von links nach rechts mit $\alpha_1, \ldots, \alpha_r \in V \cup T \cup \{\varepsilon\}$ beschriftet, so muss $A \to \alpha_1 \ldots \alpha_r$ eine Ableitungsregel der Grammatik sein. Das Zeichen ε wird nur bei Regeln vom Typ $A \to \varepsilon$ benutzt. Hier haben wir ausgenutzt, dass bei kontextfreien Grammatiken die linken Seiten der Ableitungsregeln aus genau einem Nichtterminal bestehen. Mit Hilfe eines solchen Syntaxbaumes kann man nun Ableitungen von Wörtern einer kontextfreien Sprache darstellen. Insbesondere ist es für den Bau von Compilern interessant, zu einem gegebenen Wort einen Syntaxbaum zu berechnen.

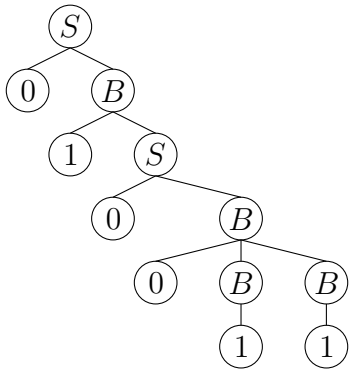
Betrachten wir als Beispiel die kontextfreie Grammatik $G = (\Sigma, V, S, P)$ mit $\Sigma = \{0, 1\}$, $V = \{S, A, B\}$ und den folgenden Ableitungsregeln:

$$S \to 0B, 1A \qquad A \to 0, 0S, 1AA \qquad B \to 1, 1S, 0BB.$$

Es ist eine interessante Übungsaufgabe, sich zu überlegen, dass diese Grammatik die Sprache $L = \{w \in \{0,1\}^* \mid |w|_0 = |w|_1 \geq 1\}$ erzeugt. Wir möchten hier aber nur als Beispiel einen Syntaxbaum für das Wort 010011 angeben, das wie folgt abgeleitet werden kann:

$$S \to 0B \to 01S \to 010B \to 0100BB \to 01001B \to 010011.$$

Der zugehörige Syntaxbaum sieht wie folgt aus:



Zu einer gegebenen Ableitung eines Wortes ist der zugehörige Syntaxbaum eindeutig bestimmt. Andersherum bestimmt der Syntaxbaum aber nicht in welcher Reihenfolge die Variablen abgeleitet werden. Zu einem Syntaxbaum gibt es also unter Umständen mehrere Ableitungen. Es gibt jedoch stets nur eine *Linksableitung*. Das ist eine Ableitung, in der in jedem Schritt die linkeste Variable abgeleitet wird.

5.4.1 Chomsky-Normalform und der Cocke-Younger-Kasami-Algorithmus

Der Algorithmus zur Lösung des Wortproblems für kontextfreie Sprachen, den wir in diesem Abschnitt angeben werden, setzt voraus, dass alle Ableitungsregeln der Grammatik eine bestimmte Form haben.

Definition 5.4.1. *Eine kontextfreie Grammatik ist in* Chomsky-Normalform*, wenn alle Ableitungsregeln die Form $A \to BC$ oder $A \to a$ für $A, B, C \in V$ und $a \in \Sigma$ haben.*

Wir werden als erstes zeigen, dass wir jede kontextfreie Grammatik G mit $\varepsilon \notin L(G)$ in Chomsky-Normalform bringen können ohne die von ihr beschriebene Sprache zu verändern. Die Tatsache, dass Grammatiken in Chomsky-Normalform das leere Wort nicht erzeugen können, folgt direkt aus den erlaubten Regeln. Dies stellt für praktische Zwecke jedoch keine wirkliche Einschränkung dar, da der Spezialfall des leeren Wortes immer gesondert betrachtet werden kann.

Sei nun also eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$ mit $\varepsilon \notin L(G)$ gegeben. Wir werden diese in mehreren Schritten in eine kontextfreie Grammatik in Chomsky-Normalform umbauen. Dabei werden wir darauf achten, dass wir die Größe der Grammatik nicht zu stark vergrößern. Formal definieren wir die Größe $s(G)$ einer Grammatik G als die Zahl der Variablen und Terminale in allen Regeln.

Der folgende Algorithmus besteht aus vier Schritten. Im ersten Schritt wird die gegebene Grammatik G in eine Grammatik G_1 überführt. Im zweiten Schritt wird G_1 in eine neue Grammatik G_2 überführt und so weiter. Wir werden diese Indizes jedoch oft weglassen, um die Notation zu vereinfachen. Das bedeutet, wenn wir in Schritt i über die Grammatik $G = (\Sigma, V, S, P)$ sprechen, so bezeichnen wir damit die Grammatik G_{i-1}, die in Schritt $i-1$ erzeugt wurde. Mit G bezeichnen wir also nicht unbedingt die gegebene Grammatik, sondern die aktuelle Grammatik, die durch die Ausführung der vorherigen Schritte entstanden ist. Außerdem werden wir mit s stets die Größe $s(G)$ der am Anfang gegebenen Grammatik G bezeichnen.

1. Schritt: (Entfernung von Terminalen bei rechten Seiten der Länge mindestens zwei)
Nach dem ersten Schritt soll es nur noch Regeln der Form $A \to v$ geben, wobei $v \in V^+$ oder $v \in \Sigma$ ist. Das heißt entweder wird A auf genau ein Terminal abgeleitet oder auf eine Folge von Variablen. Dies ist einfach zu erreichen: Für jedes $a \in \Sigma$ führen wir eine neue Variable Y_a ein, deren einzige Ableitungsregel $Y_a \to a$ ist. Nun ersetzen wir jedes Vorkommen von a in den rechten Seiten der Regeln in P durch Y_a. Dann haben alle Regeln die gewünschte Form und die Sprache, die erzeugt wird, ändert sich nicht.

Wenn wir die Variable Y_a nur für solche $a \in \Sigma$ einführen, für die das Zeichen a in mindestens einer Regel aus P vorkommt, dann vergrößern wir die Grammatik in diesem Schritt um höchstens einen konstanten Faktor. Die Größe der neuen Grammatik G_1 nach dem ersten Schritt ist also $O(s)$.

2. Schritt: (Entfernung zu langer rechter Seiten)
Nach dem ersten Schritt sind die Regeln $Y_a \to a$ die einzigen Regeln in G, die Terminale erzeugen. Diese Regeln bleiben im Folgenden unverändert und wir betrachten nur noch die anderen Regeln. Diese haben alle die Form $A \to \varepsilon$ oder $A \to B_1 \dots B_m$ für $A, B_1, \dots, B_m \in V$ und $m \geq 1$. Wir möchten erreichen, dass für jede dieser Regeln $m = 2$ gilt. In diesem zweiten Schritt werden wir nun aber zunächst nur Regeln eliminieren, für die $m \geq 3$ gilt.

Für jede solche Regel mit $m \geq 3$ führen wir $m-2$ neue Variablen $C_1, \ldots, C_{m-2}$ ein und ersetzen die Regel $A \to B_1 \ldots B_m$ durch die Regeln $A \to B_1 C_1$, $C_i \to B_{i+1} C_{i+1}$ für $i \in \{1, \ldots, m-3\}$ und $C_{m-2} \to B_{m-1} B_m$. Dabei ist zu beachten, dass wir für jede zu ersetzende Regel andere Variablen C_i hinzufügen. Da sich aus C_1 nur $B_2 \ldots B_m$ ableiten lässt, ändern wir durch diese Ersetzung die Sprache nicht. Da wir eine Regel mit insgesamt $m+1$ Variablen durch $m-1$ Regeln mit jeweils drei Variablen ersetzt haben, haben wir die Größe der Grammatik höchstens verdreifacht. Also gilt auch für die Grammatik G_2, die wir nach diesem Schritt erhalten, $s(G_2) = O(s(G_1)) = O(s)$.

Haben wir alle Regeln mit $m \geq 3$ auf diese Weise ersetzt, so gibt es insgesamt in der Grammatik G nur noch Regeln der Form $A \to B_1$, $A \to B_1 B_2$, $A \to a$ und $A \to \varepsilon$ für $A, B_1, B_2 \in V$ und $a \in \Sigma$.

3. Schritt: (Entfernung von Regeln der Form $A \to \varepsilon$)
In diesem Schritt eliminieren wir Regeln der Form $A \to \varepsilon$. Die erste Etappe dieses Schrittes besteht in der Bestimmung aller Variablen $A \in V$, für die es eine Ableitung $A \to \ldots \to \varepsilon$ gibt. Diese Menge nennen wir $W \subseteq V$, und um sie zu bestimmen, berechnen wir zunächst die Menge $W_1 \subseteq V$ mit

$$W_1 = \{A \in V \mid (A \to \varepsilon) \in P\}.$$

Für $i \geq 2$ berechnen wir

$$W_i = W_{i-1} \cup \{A \in V \mid (A \to \alpha) \in P \text{ und } \alpha \in W_{i-1}^*\}$$

solange bis das erste Mal $W_{i+1} = W_i$ gilt. Dies muss spätestens für $i = |V|$ der Fall sein, da $W_{|V|} = V$ gilt, wenn W_i in jedem Schritt vorher um mindestens ein Element gewachsen ist. Die Menge W, die wir suchen, ist dann die Menge $W_i = W_{i+1} = W_{i+2} = W_{i+3} = \ldots$, bei der wir gestoppt haben. Man kann induktiv beweisen, dass W_i genau diejenigen Variablen enthält, für die es eine Ableitung auf ε gibt, deren Syntaxbaum höchstens Tiefe i hat. Diese einfache Übung überlassen wir hier dem Leser.

Nun, da wir die Menge W kennen, führen wir die folgende Transformation durch. Zunächst streichen wir alle Regeln der Form $A \to \varepsilon$. Diese Streichung kann zu einer Veränderung der Sprache führen, was wir durch die Hinzunahme anderer Regeln kompensieren müssen. Für alle Regeln der Form $A \to BC$ mit $A, B, C \in V$ fügen wir die Regel $A \to B$ hinzu, falls $C \in W$ gilt, und wir fügen die Regel $A \to C$ hinzu, falls $B \in W$ gilt.

Mit diesen neuen Regeln können wir keine neuen Wörter erzeugen, denn gilt zum Beispiel $C \in W$, so gibt es in G eine Ableitung $A \to BC \to \ldots \to B$, welche wir nun mit den neuen Regeln lediglich direkt in einem Schritt erlauben. Auf der anderen Seite können wir mit den neuen Regeln aber immer noch alle Wörter ableiten, die wir in G ableiten können. Um dies einzusehen, betrachten wir ein beliebiges Wort $w \in L(G)$ und eine Ableitung dieses Wortes in der Grammatik G. Zu dieser Ableitung gehört ein Syntaxbaum, der unter Umständen ε-Blätter enthält. Solange dies der Fall ist, wählen wir einen Knoten v im Syntaxbaum mit der folgenden Eigenschaft aus: Alle Blätter im Teilbaum mit Wurzel v sind ε-Blätter, in dem Teilbaum unterhalb des Vaters u von v im Syntaxbaum gibt es jedoch mindestens ein Blatt, das nicht mit ε beschriftet ist. Ein solcher Knoten v existiert wegen $w \neq \varepsilon$. Ist v zum Beispiel der linke Sohn von u und sind u und v mit $A \in V$ bzw. $B \in V$ beschriftet, so wurde in der Ableitung eine Regel der Form $A \to BC$ angewendet. Von B gibt es aber eine Ableitung zu ε, weshalb $B \in W$ gilt. Wir haben somit die Regel $A \to C$ eingefügt. Wir können

nun in der Ableitung diese neue Regel anwenden, was gleichbedeutend damit ist, den Teilbaum mit Wurzel v komplett zu streichen. Diese Operation führen wir solange durch, bis es keine ε-Blätter mehr gibt. Dann haben wir eine Ableitung von w in der neuen Grammatik G_3 gefunden.

Da wir für jede Regel der Form $A \to BC$ höchstens die Regeln $A \to B$ und $A \to C$ hinzugefügt haben, wächst die Grammatik auch in diesem Schritt um höchstens einen konstanten Faktor und es gilt $s(G_3) = O(s(G_2)) = O(s(G_1)) = O(s)$.

4. Schritt: (Entfernung von Kettenregeln der Form $A \to B$)
Der letzte Schritt des Algorithmus besteht nun darin Kettenregeln der Form $A \to B$ mit $A, B \in V$ zu entfernen. Dazu betrachten wir den Graphen dessen Knoten den Variablen aus V entsprechen und der eine Kante von $A \in V$ nach $B \in V$ genau dann enthält, wenn die Regel $A \to B$ in P enthalten ist. Finden wir in diesem Graphen einen Kreis

$$A_1 \to A_2 \to \ldots \to A_r \to A_1,$$

so sind die Variablen $A_1, \ldots, A_r$ äquivalent und man kann aus jeder von ihnen exakt die gleichen Wörter ableiten. Deshalb können wir alle Variablen $A_2, \ldots, A_r$ durch A_1 ersetzen ohne die Sprache zu ändern. Einen solchen Kreis kann man zum Beispiel mit Hilfe einer Tiefensuche finden, falls er existiert. Da mit der Auflösung jedes Kreises mindestens eine Variable wegfällt, erhalten wir nach der Beseitigung von höchstens $|V|$ Kreisen einen kreisfreien Graphen auf den übriggebliebenen Variablen.

Da der Graph nun kreisfrei ist, können wir seine Knoten topologisch sortieren. Das bedeutet, wir können die übriggebliebenen Variablen so mit $A_1, \ldots, A_m$ nummerieren, dass $i < j$ für jede Kante von A_i nach A_j gilt. Nun gehen wir die Variablen in der Reihenfolge $A_{m-1}, A_{m-2}, \ldots, A_1$ durch. Wenn wir zu Variable A_k kommen, dann sind alle Regeln der Form $A_\ell \to \alpha$ mit $\ell > k$ keine Kettenregeln mehr und wir fügen für jedes α, für das es Regeln $A_k \to A_\ell$ und $A_\ell \to \alpha$ mit $\ell > k$ gibt, die Regel $A_k \to \alpha$ zur Grammatik hinzu. Haben wir das für alle α getan, dann löschen wir alle Kettenregeln der Form $A_k \to A_\ell$. Danach existieren keine Kettenregeln $A_i \to A_j$ mit $i \geq k$ mehr. Mit den neuen Regeln können wir keine neuen Wörter ableiten, da für die Regel $A_k \to \alpha$ auch schon vorher eine Ableitung $A_k \to A_\ell \to \alpha$ existierte. Auch können wir alle Wörter noch ableiten, die wir vorher ableiten konnten, denn die Ableitung $A_k \to A_\ell$ musste ja irgendwann durch ein $A_\ell \to \alpha$ fortgesetzt werden.

Dieser letzte Schritt kann die Größe der Grammatik deutlich erhöhen, da wir jede Regel $A_k \to \alpha$ maximal $|V|$ oft kopieren. Damit gilt $s(G_4) = O(|V| \cdot s(G_3)) = O(s(G_3)^2) = O(s^2)$.

Aus den Überlegungen, die wir bei der Beschreibung des Algorithmus angestellt haben, folgt direkt das folgende Ergebnis.

Theorem 5.4.2. *Der oben vorgestellte Algorithmus berechnet zu einer beliebigen kontextfreien Grammatik $G = (\Sigma, V, S, P)$ mit $\varepsilon \notin L(G)$ eine Grammatik G' in Chomsky-Normalform mit $L(G) = L(G')$ und $s(G') = O(s(G)^2)$.*

Grammatiken in Chomsky-Normalform haben eine übersichtliche Struktur, die es uns ermöglicht, das Wortproblem effizient zu lösen. Der Algorithmus, den wir vorstellen werden, heißt *Cocke-Younger-Kasami-Algorithmus* oder einfach *CYK-Algorithmus*.

Theorem 5.4.3. *Es gibt einen Algorithmus, der in Zeit $O(|P|n^3)$ für eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$ in Chomsky-Normalform und ein Wort $w \in \Sigma^*$ mit $|w| = n$ entscheidet, ob $w \in L(G)$ gilt.*

Beweis. Der CYK-Algorithmus basiert auf der Methode der dynamischen Programmierung, die wir bereits mehrfach in der Vorlesung angewendet haben. Das bedeutet, das Gesamtproblem, zu entscheiden, ob $w \in L(G)$ gilt, wird in kleinere Teilprobleme zerlegt, deren Lösungen zu einer Lösung des Gesamtproblems zusammengesetzt werden. Für jedes Paar von Indizes i und j mit $1 \leq i \leq j \leq n$ bezeichnen wir mit $V_{ij} \subseteq V$ die Menge aller Variablen $A \in V$, für die es eine Ableitung $A \to \ldots \to w_i \ldots w_j$ gibt. Wir möchten entscheiden, ob das Startsymbol zu der Menge V_{1n} gehört, ob es also eine Ableitung $S \to \ldots \to w_1 \ldots w_n = w$ gibt.

Wir berechnen die Mengen V_{ij} nacheinander sortiert nach wachsender Differenz $\ell = j - i$. Das heißt, wir fangen zunächst mit allen Mengen an, für die $\ell = 0$ gilt. Dies sind die Mengen V_{ii} für $i \in \{1, \ldots, n\}$. Da G in Chomsky-Normalform ist, besteht die Menge V_{ii} aus genau denjenigen Variablen $A \in V$, für die es die Regel $A \to w_i$ in P gibt. Somit können wir jedes V_{ii} in Zeit $O(|P|)$ berechnen.

Sei nun $\ell > 0$ und seien alle V_{ij} für $j - i < \ell$ bereits berechnet. Nun betrachten wir nacheinander alle V_{ij} mit $i - j = \ell$. Für $A \in V_{ij}$ gibt es wegen der Chomsky-Normalform und $\ell > 0$ eine Ableitung der Form $A \to BC \to \ldots \to w_i \ldots w_j$. Das bedeutet auch, es muss einen Index $k \in \{i, \ldots, j-1\}$ und Ableitungen $B \to \ldots \to w_i \ldots w_k$ und $C \to \ldots \to w_{k+1} \ldots w_j$ geben. Dies ist die zentrale Beobachtung, die wir zur Berechnung von V_{ij} benötigen. Es gilt nämlich

$$V_{ij} = \{A \in V \mid \exists k \in \{i, \ldots, j-1\} : \exists B, C \in V : (A \to BC) \in P \text{ und } B \in V_{ik} \text{ und } C \in V_{(k+1)j}\}.$$

Wegen $k - i < \ell$ und $j - (k+1) < \ell$ sind die Mengen V_{ik} und $V_{(k+1)j}$ bereits bekannt. Wir können V_{ij} also für jedes Paar mit $j - i = \ell$ in Zeit $O(n|P|)$ berechnen, indem wir alle möglichen Stellen $k \in \{i, \ldots, j-1\}$ und alle Regeln $(A \to BC) \in P$ durchgehen.

Da es $O(n^2)$ viele Paare i und j mit $1 \leq i \leq j \leq n$ gibt und wir jedes V_{ij} in Zeit $O(n|P|)$ berechnen können, ergibt sich insgesamt eine Laufzeit von $O(n^3|P|)$ für die Berechnung von V_{1n}. Es gilt $w \in L(G)$ genau dann, wenn $S \in V_{1n}$. □

Der CYK-Algorithmus hat für eine feste Menge von Ableitungsregeln eine kubische Laufzeit in der Länge des Wortes. Eine solche Laufzeit ist zwar in der Theorie effizient, sie kann aber bei langen Quelltexten in der Praxis zu Problemen führen. Aus diesem Grunde werden bei dem Entwurf von Programmiersprachen Unterklassen von kontextfreien Grammatiken betrachtet, die noch mächtig genug sind, um gängige Programmiersprachen zu erzeugen, für die das Wortproblem aber in linearer Laufzeit gelöst werden kann. Bei diesen Sprachen handelt es sich um *deterministisch kontextfreie Sprachen*. Was das bedeutet, werden wir im weiteren Verlauf der Vorlesung kurz diskutieren.

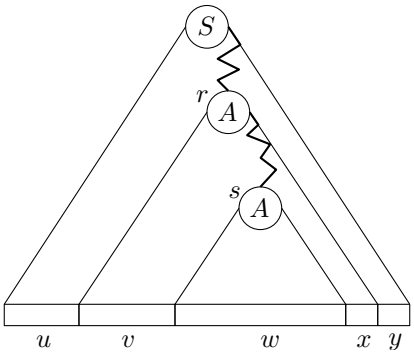
5.4.2 Pumping-Lemma für kontextfreie Sprachen

Nun möchten wir uns aber zunächst damit beschäftigen, welche Sprachen überhaupt durch kontextfreie Grammatiken beschrieben werden können. Ähnlich wie bei regulären Sprachen werden wir auch hier eine Variante des Pumping-Lemmas herleiten, mit deren Hilfe man für gewisse Sprachen nachweisen kann, dass sie nicht kontextfrei sind.

Lemma 5.4.4 (Pumping-Lemma)**.** *Sei L eine kontextfreie Sprache mit $\varepsilon \notin L$. Dann gibt es eine Konstante n, sodass für alle $z \in L$ mit $|z| \geq n$ gilt: Das Wort z kann in fünf Teile $z = uvwxy$ mit $|vx| \geq 1$ und $|vwx| \leq n$ zerlegt werden, sodass $uv^iwx^iy \in L$ für alle $i \geq 0$ gilt.*

Beweis. Bei dem Beweis des Pumping-Lemmas für reguläre Sprachen haben wir ausgenutzt, dass eine Sprache genau dann regulär ist, wenn es einen DFA für sie gibt. Bei kontextfreien Sprachen kennen wir noch keine solche Charakterisierung über ein Rechnermodell, wir können aber die Chomsky-Normalform ausnutzen. Sei dazu $G = (\Sigma, V, S, P)$ eine kontextfreie Grammatik für die Sprache L. Der zu einer Ableitung gehörige Syntaxbaum ist wegen der Chomsky-Normalform ein Baum, in dem jeder innere Knoten entweder genau zwei Söhne hat, die keine Blätter sind, oder genau einen Sohn, der ein Blatt ist. Hat ein solcher Baum mindestens 2^k Blätter, so muss er Höhe mindestens $k + 1$ haben.

Wir setzen nun $k = |V|$ und $n = 2^k$. Sei $z \in L$ ein beliebiges Wort mit $|z| \geq n$. Dann muss jeder zu z gehörige Syntaxbaum Höhe mindestens $|V| + 1$ haben. Wir betrachten ein Blatt, das maximale Entfernung zur Wurzel hat, und bezeichnen mit W einen Weg von der Wurzel des Syntaxbaums zu diesem Blatt. Dieser Weg W enthält mindestens $|V|+1$ Kanten. Entlang dieses Weges gibt es mindestens $|V|+2$ Knoten, von denen nur der letzte mit einem Terminalzeichen aus Σ beschriftet ist. Die anderen mindestens $|V| + 1$ Knoten sind mit Variablen aus V beschriftet, von denen also mindestens eine doppelt vorkommen muss. Wir betrachten nun die untersten beiden Knoten auf dem Pfad W, die mit der gleichen Variable beschriftet sind. Wir nennen diese Knoten r und s und die Variable $A \in V$, und wir gehen davon aus, dass r der höhere der beiden Knoten ist. Die Anzahl Kanten auf dem Teilpfad von r zum Blatt ist dann gemäß obiger Argumentation höchstens $|V| + 1$. Wir befinden uns dann in der Situation, die in folgender Abbildung dargestellt ist.



In obiger Abbildung haben wir das Wort z bereits in fünf Teile zerlegt: vwx ist der Teil des Wortes z, der unterhalb von Knoten r im Syntaxbaum abgeleitet wird, und w ist der Teil, der unterhalb von s abgeleitet wird. Dadurch ergeben sich auch eindeutig die anderen Teile. Da gemäß der Wahl von W der Teilbaum mit Wurzel r Höhe höchstens $|V| + 1$ hat, gilt $|vwx| \leq 2^{|V|} = n$. Des Weiteren repräsentiert der Knoten r mit seinen beiden Söhnen eine Ableitungsregel der Form $A \to BC$, wobei entweder B oder C nicht auf dem Weg W liegt. Da keine der Variablen zu ε abgeleitet werden kann, impliziert dies sofort, dass $|vx| \geq 1$ gelten muss.

Damit sind die Voraussetzungen an die Zerlegung erfüllt und wir müssen uns nur noch überlegen, dass für jedes $i \geq 0$ das Wort uv^iwx^iy mit der Grammatik G abgeleitet werden kann. Der Syntaxbaum sagt uns, dass folgende Ableitungen möglich sind:

$$S \to \ldots \to uAy \qquad A \to \ldots \to vAx \qquad A \to \ldots \to w.$$

Damit erhalten wir sofort für $i = 0$ die Ableitung

$$S \to \ldots \to uAy \to \ldots \to uwy = uv^0wx^0y$$

und für $i \geq 1$ die Ableitung

$$S \to \ldots \to uAy \to \ldots \to uvAxy \to \ldots \to uv^2Ax^2y \to \ldots \to uv^iAx^iy \to \ldots \to uv^iwx^iy.$$

Damit ist das Pumping-Lemma für kontextfreie Sprachen bewiesen. □

Genau wie für das Pumping-Lemma für reguläre Sprachen interessiert uns auch hier für die Anwendung die Negation der im Lemma formulierten Aussage:

$$\begin{aligned}
&\forall n \in \mathbb{N}: \\
&\quad \exists z \in L, |z| \geq n: \\
&\quad\quad \forall \text{ Zerlegung } z = uvwxy, |vwx| \leq n, |vx| \geq 1: \\
&\quad\quad\quad \exists i \geq 0 : uv^iwx^iy \notin L: \\
&\Rightarrow \text{L ist nicht kontextfrei}
\end{aligned}$$

Wir stellen uns das wieder als ein Spiel vor, in dem Peter (Prover) Vera (Verifier) davon überzeugen möchte, dass die negierte Aussage gilt. Vera möchte sich aber nur davon überzeugen lassen, wenn die Aussage wirklich wahr ist. Vera übernimmt die ∀-Rolle und Peter die ∃-Rolle. Das Spiel sieht wie folgt aus:

1. Runde: Vera wählt ein $n \in \mathbb{N}$.
2. Runde: Peter wählt ein $z \in L$ mit $|z| \geq n$.
3. Runde: Vera wählt eine Zerlegung $z = uvwxy$ mit $|vwx| \leq n$ und $|vx| \geq 1$.
4. Runde: Peter wählt ein $i \geq 0$.

Gelingt es Peter, in der 4. Runde ein i so zu wählen, dass $uv^iwx^iy \notin L$ gilt, so hat er gewonnen. Hat er eine Strategie, auf jede Wahl, die Vera in den Runden 1 und 3 trifft, zu reagieren, und schafft es am Ende immer, ein i zu wählen, für das $uv^iwx^iy \notin L$ gilt, so gilt die Negation des Pumping-Lemmas und die Sprache ist nicht kontextfrei.

Wir betrachten die Sprache $L = \{a^ib^ic^i \mid i \in \mathbb{N}\}$ als Beispiel. Wir wollen zeigen, dass L nicht kontextfrei ist und übernehmen Peters Rolle.

1. Runde: Vera wählt ein beliebiges $n \in \mathbb{N}$.
2. Runde: Wir wählen $z = a^nb^nc^n$. Offenbar gilt $z \in L$ und $|z| \geq n$.
3. Runde: Vera wählt eine beliebige Zerlegung $z = uvwxy = a^nb^nc^n$, wobei $|vwx| \leq n$ und $|vx| \geq 1$ gilt.
4.Runde: Da vwx höchstens die Länge n haben darf, können in diesem Teil höchstens zwei verschiedene Buchstaben vorkommen. Wir wählen nun $i = 2$.

Betrachten wir das Wort uv^2wx^2y. Aus unseren Vorüberlegungen folgt, dass die verdoppelten Teile v und x höchstens zwei verschiedene Buchstaben enthalten. Wegen der Annahme an die Zerlegung erhalten sie aber auch mindestens einen Buchstaben. In dem Wort uv^2wx^2y gibt es also entweder einen oder zwei aber nicht drei Buchstaben, die öfter vorkommen als im Wort $uvwxy$. Demzufolge kann es nicht sein, dass alle drei Buchstaben im Wort uv^2wx^2y gleich oft auftreten. Also ist $uv^2wx^2y \notin L$. Wir haben uns also auf ein beliebiges n und eine beliebige Zerlegung eingestellt und in jedem Fall ein Wort uv^iwx^iy gefunden, das nicht zur Sprache gehört. Also haben wir gewonnen und somit ist die Sprache nicht kontextfrei.

5.5 Kellerautomaten und Syntaxanalyse

Wir möchten nun ein Rechnermodell einführen, das in der Lage ist, genau die kontextfreien Sprachen zu entscheiden. Ein solches Rechnermodell ist hilfreich, da es einen tieferen Einblick in die Klasse der kontextfreien Sprachen gibt. Es wird uns insbesondere dabei helfen, eine Teilklasse der kontextfreien Sprachen zu identifizieren, die noch mächtig genug ist, um gängige Programmiersprachen zu beschreiben, für die das Wortproblem aber in Linearzeit gelöst werden kann. Wir wissen bereits, dass endliche Automaten nicht mächtig genug sind, die Klasse der kontextfreien Sprachen zu entscheiden. Um ein mächtigeres Rechnermodell zu erhalten, werden wir endliche Automaten mit einem unbegrenzten Stack als Speicher erweitern. Wegen der begrenzten Zeit, die uns in der Vorlesung noch zur Verfügung steht, werden wir in diesem Kapitel viele Beweise und Details weglassen und nur versuchen, einen Überblick über die wesentlichen Konzepte zu vermitteln.

Bevor wir das Rechnermodell formal definieren, benötigen wir eine weitere Normalform für kontextfreie Sprachen.

Definition 5.5.1. *Eine kontextfreie Grammatik ist in* Greibach-Normalform*, wenn alle Ableitungsregeln die Form $A \to a\alpha$ für $A \in V$, $a \in \Sigma$ und $\alpha \in V^*$ haben.*

Es ist möglich, jede kontextfreie Grammatik in Chomsky-Normalform in eine Grammatik in Greibach-Normalform für dieselbe Sprache zu überführen. Diese Transformation werden wir hier allerdings nicht beschreiben. Stattdessen betrachten wir, wie uns die Greibach-Normalform dabei hilft, ein Rechnermodell für kontextfreie Sprachen zu finden. Wenn wir uns eine Linksableitung eines Wortes $w_1 \dots w_n$ in einer kontextfreien Grammatik in Greibach-Normalform anschauen, dann stellen wir fest, dass nach i Schritten eine Zeichenfolge der Form $w_1 \dots w_i X_1 \dots X_r$ erzeugt wurde, wobei $X_1, \dots, X_r \in V$ Variablen sind. Für $i < n$ gilt $r \geq 1$, und für $i = n$ gilt $r = 0$. In Schritt $i + 1$ der Linksableitung wird dann die Variable X_1 zu $w_{i+1} Y_1 \dots Y_{r'}$ abgeleitet und wir erhalten insgesamt die Zeichenfolge $w_1 \dots w_{i+1} Y_1 \dots Y_{r'} X_2 \dots X_r$.

Basierend auf diesen Beobachtungen konstruieren wir nun einen *nichtdeterministischen Kellerautomaten*, der für ein Wort w und eine kontextfreie Grammatik G in Greibach-Normalform entscheidet, ob $w \in L(G)$ gilt. Im Gegensatz zu endlichen Automaten besitzt dieser Kellerautomat einen unendlichen Speicher, den er aber nur in Form eines Stacks (früher auf Deutsch auch „Keller“) benutzen darf. Das bedeutet, der unendliche Speicher arbeitet ausschließlich nach dem LIFO-Prinzip (Last In – First Out). Der Kellerautomat liest das Eingabewort w genau wie ein endlicher Automat einmal Zeichen für Zeichen von links nach rechts. Beim Lesen versucht er Schritt für Schritt eine Linksableitung für w zu konstruieren. Dazu initialisiert er seinen Speicher, indem er vor dem Lesen von w das Startsymbol auf den Stack schreibt. Liest er nun ein Zeichen w_i des Eingabewortes, so entfernt er die zuletzt hinzugefügte Variable $A \in V$ vom Stack und wählt nichtdeterministisch eine Regel der Form $A \to w_i Y_1 \dots Y_{r'}$ aus. Gibt es keine solche Regel, so verwirft der Automat das Wort w. Ansonsten schreibt er die Variablen in der Reihenfolge $Y_{r'}, \dots, Y_1$ auf den Stack und fährt mit dem nächsten Zeichen w_{i+1} analog fort. Der Automat akzeptiert das Wort w dann, wenn der Stack nach dem Lesen des letzten Zeichens w_n leer ist.

Wir weisen noch einmal darauf hin, dass der Automat nichtdeterministisch arbeitet. Hat er in einem Schritt mehrere Regeln der Form $A \to w_i Y_1 \dots Y_{r'}$ zur Auswahl, so ist nicht bestimmt, welche davon er anwendet. Genau wie bei nichtdeterministischen endlichen Automaten definieren wir auch hier wieder, dass der Automat das Wort w akzeptiert, wenn es

mindestens einen gültigen Rechenweg, d. h. eine mögliche Auswahl von Regeln, gibt, für die der Automat w akzeptiert.

Sei als Beispiel die kontextfreie Grammatik $G = (\Sigma, V, S, P)$ mit $\Sigma = \{0, 1\}$, $V = \{S, A, B, C\}$ und den Regeln

$$S \to 0S, 0A \qquad A \to 0BC \qquad B \to 1 \qquad C \to 0$$

gegeben. Wir betrachten die folgende Linksableitung für das Wort 00010:

$$S \to 0S \to 00A \to 000BC \to 0001C \to 00010.$$

Liest der nichtdeterministische Kellerautomat nun das Wort 00010 Zeichen für Zeichen, so durchläuft er die folgende Sequenz von Konfigurationen:

Schritt	bereits gelesen	noch zu lesen	Stack	Regel
1	ε	00010	S	$S \to 0S$
2	0	0010	S	$S \to 0A$
3	00	010	A	$A \to 0BC$
4	000	10	BC	$B \to 1$
5	0001	0	C	$C \to 0$
6	00010	ε	$\emptyset$	—

In diesem Beispiel hat der Automat richtig geraten: Obwohl in den Schritten 1 und 2 das gleiche Zeichen (nämlich eine 0) gelesen wurde und das oberste Zeichen auf dem Stack das gleiche war (nämlich S) hat der Automat in Schritt 1 die Regel $S \to 0S$ und in Schritt 2 die Regel $S \to 0A$ angewendet. Hätte er bei einem dieser Schritte eine andere Regel angewendet, so hätte er nicht mit einem leeren Stack nach dem Lesen des gesamten Wortes terminieren können.

5.5.1 Nichtdeterministische Kellerautomaten

Für die weiteren Betrachtungen ist der ungewohnte Akzeptanzmodus „leerer Stack“ unhandlich. Uns wäre lieber, wenn der Automat beim Lesen eines Wortes verschiedene Zustände durchläuft und wir wieder einige dieser Zustände als akzeptierend auszeichnen könnten. Aus diesem Grunde weicht die formale Definition von Kellerautomaten etwas von dem ab, was man nach der informellen obigen Beschreibung erwarten könnte.

Definition 5.5.2. *Ein* nichtdeterministischer Kellerautomat *(nondeterministic pushdown automaton, NPDA) M besteht aus sieben Komponenten $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ mit den folgenden Bedeutungen:*

- *Q ist eine endliche Menge, die* Zustandsmenge.
- *Σ ist eine endliche Menge, das* Eingabealphabet.
- *Γ ist eine endliche Menge, das* Stackalphabet.
- *δ ist die* Zustandsüberführungsfunktion. *Sie bildet $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ in die Potenzmenge von $Q \times \Gamma^*$ ab.*
- *$q_0 \in Q$ ist der* Startzustand.
- *$Z_0 \in \Gamma$ ist die* Initialisierung des Stacks.
- *$F \subseteq Q$ ist die Menge der* akzeptierenden Zustände.

Zu jedem Zeitpunkt lässt sich die Konfiguration eines Kellerautomaten durch ein Tripel (q, w, γ) beschreiben, wobei $q \in Q$ der aktuelle Zustand ist, $w \in \Sigma^*$ der Teil der Eingabe, der noch nicht gelesen wurde, und $\gamma \in \Gamma^*$ der aktuelle Stackinhalt. Ist die aktuelle Konfiguration $(q, w_1 \dots w_k, Z_1 \dots Z_\ell)$ und ist $(q', Z'_1 \dots Z'_j) \in \delta(q, w_1, Z_1)$, so kann der Automat in die Konfiguration $(q', w_2 \dots w_k, Z'_1 \dots Z'_j Z_2 \dots Z_\ell)$ wechseln. Die Definition der Zustandsüberführungsfunktion erlaubt es auch, dass der Automat sogenannte *ε-Bewegungen* macht; dabei handelt es sich um Zustandsübergänge bei denen kein Eingabezeichen gelesen wird. Ist $(q', Z'_1 \dots Z'_j) \in \delta(q, \varepsilon, Z_1)$, so kann der Automat aus der aktuellen Konfiguration in die Konfiguration $(q', w_1 \dots w_k, Z'_1 \dots Z'_j Z_2 \dots Z_\ell)$ wechseln.

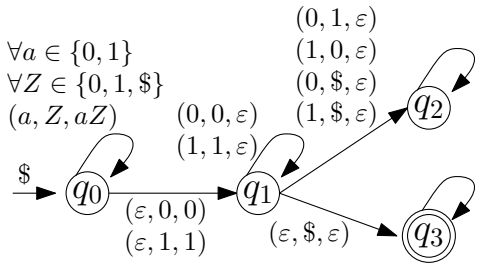
Die beiden Möglichkeiten, das Akzeptanzverhalten eines solchen Kellerautomaten zu definieren, sehen wie folgt aus:

- *Akzeptanz durch leeren Stack:* Ein NPDA M akzeptiert das Wort $w \in \Sigma^*$ genau dann, wenn es einen zulässigen Rechenweg gibt, der von der Anfangskonfiguration (q_0, w, Z_0) in eine Konfiguration $(q, \varepsilon, \varepsilon)$ mit $q \in Q$ führt.
- *Akzeptanz durch akzeptierende Zustände:* Ein NPDA M akzeptiert das Wort $w \in \Sigma^*$ genau dann, wenn es einen zulässigen Rechenweg gibt, der von der Anfangskonfiguration (q_0, w, Z_0) in eine Konfiguration (q, ε, γ) mit $q \in F$ und $\gamma \in \Gamma^*$ führt.

Man kann zeigen, dass beide Akzeptanzmodi gleichmächtig sind. Man kann also jeden NPDA mit einem der beiden Akzeptanzmodi in einen NPDA für die gleiche Sprache mit dem anderen Akzeptanzmodus umbauen. Deshalb werden wir uns im Folgenden nur noch mit dem zweiten Akzeptanzmodus mit akzeptierenden Zuständen beschäftigen. Wir haben Kellerautomaten mit dem Ziel definiert, ein Rechnermodell für kontextfreie Sprachen zu erhalten. Aus unserer obigen Diskussion im Anschluss an die Greibach-Normalform folgt leicht, dass wir für jede kontextfreie Sprache tatsächlich einen NPDA bauen können. Es könnte aber sein, dass wir das Modell durch die Einführung einer Zustandsmenge und das Erlauben von ε-Übergängen zu mächtig gemacht haben. Das folgende Theorem, das wir in der Vorlesung nicht beweisen werden, sagt, dass dies nicht der Fall ist.

Theorem 5.5.3. *Die Klasse der kontextfreien Sprachen stimmt mit der Klasse von Sprachen überein, die von NPDAs entschieden werden können.*

Als Beispiel betrachten wir einen NPDA für die Sprache $L = \{ww^R \mid w \in \{0,1\}^+\}$. Liest der NPDA ein Wort $z = ww^R$, so sollte er in der ersten Hälfte einfach alle Zeichen, die er liest, auf den Stack schreiben. Nach der ersten Hälfte von z steht dann genau das Wort w^R auf dem Stack, welches dann beim Lesen der zweiten Hälfte mit den gelesenen Zeichen verglichen werden kann. Der NPDA muss also nichtdeterministisch entscheiden, wann die zweite Hälfte des Wortes beginnt. Wir stellen diesen NPDA wieder als Graphen darf, genauso wie wir es für endliche Automaten getan haben. Die Kanten des Graphen müssen jetzt jedoch noch weitere Informationen enthalten. In der folgenden Abbildung benutzen wir Kantenbeschriftungen der Form (a, Z, γ) mit $a \in \Sigma \cup \{\varepsilon\}$, $Z \in \Gamma = \{0, 1, \$\}$ und $\gamma \in \Gamma^*$. Zeigt eine solche Kante von Zustand q zu Zustand q', so symbolisiert sie $(q', \gamma) \in \delta(q, a, Z)$. Der in den Startzustand eingehende Pfeil ist nun zusätzlich mit Z_0, der Initialisierung des Stacks, beschriftet.



Da q_3 der einzige akzeptierende Zustand ist, akzeptiert der Automat ein Wort z genau dann, wenn es einen gültigen Rechenweg von $(q_0, z, \$)$ zu $(q_3, \varepsilon, \gamma)$ mit $\gamma \in \Gamma^*$ gibt. Der NPDA kann sich bei einem Wort $z = ww^R$ genauso verhalten wie wir es beschrieben haben: Er liest zunächst w, bleibt währenddessen im Zustand q_0 und schreibt w^R auf den Stack. Dann macht er eine ε-Bewegung in den Zustand q_1 ohne den Inhalt des Stacks zu verändern. Dort bleibt er, während er das Wort w^R liest. Zum Schluss, wenn nur noch \$ auf dem Stack steht, leert der Automat den Stack mit einer ε-Bewegung und erreicht die Konfiguration $(q_3, \varepsilon, \varepsilon)$. Es ist eine Übungsaufgabe für den Leser, sich zu überlegen, warum der Automat für ein Wort $z \notin L$ keine Konfiguration $(q_3, \varepsilon, \gamma)$ mit $\gamma \in \Gamma^*$ erreichen kann.

5.5.2 Deterministisch kontextfreie Sprachen

Genau wie bei endlichen Automaten sind nichtdeterministische Modelle wie das des NPDA konzeptuell wichtig und sie erleichtern die theoretische Betrachtung von kontextfreien Sprachen. Allerdings handelt es sich bei ihnen um keine Rechnermodelle, die man in der realen Welt direkt bauen kann. Haben wir hingegen für eine kontextfreie Sprache einen deterministischen Kellerautomaten, so könnten wir diesen einsetzen, um das Wortproblem zu lösen. Deshalb werden wir uns jetzt mit deterministischen Kellerautomaten beschäftigen. Für die Definition eines solchen Automaten erinnern wir uns, dass deterministisch bedeutet, dass zu jedem Zeitpunkt ein eindeutiger Übergang erfolgen muss. Das schließt insbesondere die ε-Bewegungen ein und führt zu folgender Definition.

Definition 5.5.4. *Ein* deterministischer Kellerautomat *(deterministic pushdown automaton, DPDA) M ist ein NPDA $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ mit der zusätzlichen Eigenschaft, dass für jedes Tripel $(q, a, Z) \in Q \times \Sigma \times \Gamma$ die folgende Aussage gilt:*

$$|\delta(q, a, Z)| + |\delta(q, \varepsilon, Z)| \leq 1.$$

In der Definition eines DPDA wird also gefordert, dass es zu jeder Konfiguration höchstens eine gültige Nachfolgekonfiguration gibt. Gemäß dieser Definition kann es somit vorkommen, dass der DPDA, ohne bereits das gesamte Wort gelesen zu haben, eine Konfiguration erreicht, für die es keine erlaubte Nachfolgekonfiguration gibt. In einem solchen Fall verwirft der DPDA, da er keine Konfiguration (q, ε, γ) mit $q \in F$ und $\gamma \in \Gamma^*$ erreichen kann.

Der Automat aus dem Beispiel für die Sprache $L = \{ww^R \mid w \in \{0,1\}^+\}$ nutzt essentiell aus, dass er bei einem Wort $z \in L$ den richtigen Moment für die ε-Bewegung erraten kann. Dies geht mit deterministischen Kellerautomaten nicht, und tatsächlich kann man auch zeigen, dass es für die Sprache L keinen DPDA gibt. Dies ist insofern bemerkenswert, als dass es zeigt, dass nichtdeterministische Kellerautomaten mächtiger sind als deterministische. Dies steht im Gegensatz zu endlichen Automaten, für die wir mit Hilfe der Potenzmengenkonstruktion gezeigt haben, dass der Nichtdeterminismus die Mächtigkeit nicht vergrößert.

Eine Sprache $L \subseteq \{0,1\}^*$ ist *deterministisch kontextfrei*, wenn es einen deterministischen Kellerautomaten gibt, der sie erkennt. Die Klasse der deterministisch kontextfreien Sprachen ist nach unseren Vorüberlegungen eine Teilmenge der kontextfreien Sprachen. Sie ist aber auch eine echte Erweiterung der regulären Sprachen, was das Beispiel der Sprache $L' = \{w\#w^R \mid w \in \{0,1\}^+\}$ zeigt. Diese Sprache ist nicht regulär, sie kann allerdings von einem DPDA erkannt werden, der ähnlich arbeitet wie der NPDA für L, den wir oben vorgestellt haben. Der Unterschied ist nur, dass der DPDA nicht erraten muss, wann die zweite Hälfte des Wortes beginnt, da es ein explizites Trennsymbol # gibt.

5.5.3 Bottom-up Syntaxanalyse

Syntaxanalyse bezeichnet das Problem, für eine gegebene kontextfreie Grammatik G und ein gegebenes Wort $w \in \Sigma^*$ zu entscheiden, ob $w \in L(G)$ gilt, und im positiven Fall einen Syntaxbaum für w in der Grammatik G zu berechnen. Dies ist eine der wesentlichen Aufgaben, die ein Compiler durchführen muss. Mit dem CYK-Algorithmus haben wir bereits einen Algorithmus kennengelernt, der dieses Problem für jede kontextfreie Grammatik löst. Wir haben nur beschrieben, wie der CYK-Algorithmus entscheidet, ob $w \in L(G)$ gilt. Der Algorithmus kann aber leicht dahingehend erweitert werden, dass er auch den Syntaxbaum berechnet. Das Problem ist nur seine relativ hohe Laufzeit, die kubisch in der Länge des Wortes ist. Dies ist in der Praxis bei langen Quelltexten nicht effizient genug. Wir haben die Klasse der deterministisch kontextfreien Sprachen mit der Absicht definiert, eine Sprachklasse zu erhalten, die mächtig genug ist, gängige Programmiersprachen zu beschreiben, für die die Syntaxanalyse aber in Linearzeit durchgeführt werden kann. In diesem Abschnitt werden wir kurz besprechen, dass dies wirklich der Fall ist.

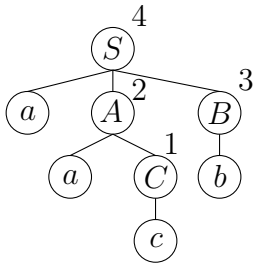
Zunächst präzisieren wir aber die Aufgabenstellung der Bottom-up Syntaxanalyse. Diese trägt ihren Namen, da sie den Syntaxbaum von unten nach oben erzeugt. Sei $w = w_1 \dots w_n$ ein Wort, das in einer kontextfreien Grammatik G abgeleitet werden kann. Dann betrachten wir eine *Rechtsableitung* von w, also eine Ableitung, in der wir stets die rechteste Variable zuerst ableiten. Bei der Syntaxanalyse berechnen wir diese Ableitung nun rückwärts. Zu Beginn kennen wir nur die n Blätter des Syntaxbaumes, die wir mit den Terminalen $w_1, \dots, w_n$ beschriften. Nun betrachten wir alle Knoten des Syntaxbaums, deren Kinder bereits alle beschriftet sind. Von diesen Knoten erzeugen wir den linkesten als nächstes und beschriften ihn mit einer Variable. Diesen Prozess wiederholen wir, bis alle Knoten des Baumes beschriftet sind. Sei beispielsweise die Grammatik G mit $\Sigma = \{a,b,c\}$, $V = \{S,A,B,C\}$ und den Regeln

$$S \to aAB \qquad A \to aC \qquad B \to b \qquad C \to c$$

gegeben. Dann ist

$$S \to aAB \to aAb \to aaCb \to aacb$$

eine Rechtsableitung mit dem folgenden Syntaxbaum.



In diese Syntaxbaum geben die Zahlen an den inneren Knoten an, in welcher Reihenfolge sie bei der Bottom-up Syntaxanalyse berechnet werden. Wir sehen in diesem Beispiel, dass dies genau die umgedrehte Reihenfolge der Rechtsableitung ist.

Der Algorithmus für die Syntaxanalyse, den wir anstreben, soll ähnlich arbeiten wie ein deterministischer Kellerautomat. Das heißt, er soll das Wort w einmal von links nach rechts lesen und während des Lesens die Regeln einer Rechtsableitung von w rückwärts ausgeben. Für allgemeine kontextfreie Grammatiken erweist sich dies jedoch als schwierig. Gibt es beispielsweise nur die Regeln $S \to aS$, $S \to b$ und $S \to c$, so kann erst beim Lesen des letzten Zeichens entschieden werden, welches die letzte Regel einer Rechtsableitung ist.

Ein weiteres Problem tritt ein, wenn es eine Regel $A \to w_1 \dots w_j$ gibt, wenn wir aber nach dem Lesen von $w_1 \dots w_j$ trotzdem nicht entscheiden können, ob $A \to w_1 \dots w_j$ die letzte Regel einer Rechtsableitung war. Gibt es zum Beispiel die Regeln $S \to AB$, $S \to CD$, $A \to a$, $B \to bc$, $C \to ac$ und $D \to b$ und ist das erste gelesene Zeichen ein a, so können wir nicht sagen, ob die letzte Regel einer Rechtsableitung $A \to a$ oder $C \to ac$ war. Um dies zu entscheiden, müssen wir bereits das zweite Zeichen gelesen haben. Dies zeigt, dass wir den Algorithmus zur Syntaxanalyse mit einem *Lookahead* ausstatten müssen. Wir bezeichnen mit k die Anzahl an Zeichen, die vorausgeschaut werden darf, und gehen davon aus, dass k eine Konstante ist. Informell bedeutet das, dass bei Existenz einer Regel $A \to w_1 \dots w_j$ spätestens nach dem Lesen von $w_1 \dots w_{j+k}$ effizient entscheidbar sein muss, ob $A \to w_1 \dots w_j$ die letzte Regel einer Rechtsableitung von w ist. Diese Anforderung ergibt einen LR(k)-Parser.

Definition 5.5.5 (Informelle Definition eines LR(k)-Parsers)**.** *Ein LR(k)-Parser ist ein Algorithmus für eine kontextfreie Grammatik G, der wie ein deterministischer Kellerautomat arbeitet und das Eingabewort w einmal von links nach rechts liest (für diese Leserichtung steht das L). Er soll eine Fehlermeldung ausgeben, wenn w nicht zur Sprache $L(G)$ gehört, und ansonsten eine Folge von Regeln ausgeben, die in umgekehrter Reihenfolge eine Rechtsableitung von w bilden (dafür steht das R). Ein LR(k)-Parser muss mit einem Lookahead von k auskommen.*

Nun stellt sich natürlich sofort die Frage, für welche Sprachen ein solcher LR(k)-Parser konstruiert werden kann. Aus Zeitgründen können wir diese Frage leider in der Vorlesung nicht mehr beantworten. Es sei nur so viel gesagt: Wenn man die Anforderungen, die an einen LR(k)-Parser gestellt werden, genau untersucht, dann kann man daraus Eigenschaften ableiten, die eine kontextfreie Grammatik erfüllen muss, damit es für sie einen LR(k)-Parser gibt. Grammatiken mit diesen Eigenschaften werden als LR(k)-Grammatiken bezeichnet. Man kann nachweisen, dass für jedes $k \geq 1$ die Klasse der Sprachen, die von LR(k)-Grammatiken erzeugt werden, mit der Klasse der deterministisch kontextfreien Sprachen übereinstimmt.

Literatur

Die Inhalte dieses Kapitels können in [1] und [2] nachgelesen werden. Dort finden sich auch zahlreiche Erweiterungen und Übungsaufgaben. Zusätzlich sind die wesentlichen Konzepte in [3] informal dargestellt.

[1] Norbert Blum. Einführung in Formale Sprachen, Berechenbarkeit, Informations- und Lerntheorie. Oldenbourg Verlag, 2006.

[2] Ingo Wegener. Theoretische Informatik – eine algorithmenorientierte Einführung. Teubner Verlag, 3. Auflage, 2005.

[3] Ingo Wegener. Kompendium Theoretische Informatik: Eine Ideensammlung. Teubner Verlag, 1996.