Skript zur Vorlesung „Theoretische Informatik 2“

Prof. Dr. Georg Schnitger

Sommersemester 2013

Hinweise auf Fehler und Anregungen zum Skript bitte an

poloczek@thi.informatik.uni-frankfurt.de

Mit einem Stern gekennzeichnete Abschnitte werden in der Vorlesung nur kurz angesprochen.

Inhaltsverzeichnis

1 Einführung **3**

1.1 Ziele der Vorlesung 3

1.2 Begriffe und Notationen 4

1.3 Das Wortproblem 7

1.4 Komplexitätsklassen 9

1.5 Literatur 13

I Formale Sprachen **15**

2 Endliche Automaten und reguläre Sprachen **17**

2.1 Nichtdeterministische endliche Automaten 23

2.1.1 Die Potenzmengenkonstruktion 25

2.2 Das Pumping-Lemma 28

2.3 Minimierung endlicher Automaten 31

2.3.1 Der Äquivalenzklassenautomat 34

2.3.2 Die Nerode-Relation 38

2.4 Abschlusseigenschaften und Entscheidungsprobleme 46

2.5 Reguläre Ausdrücke 48

2.6 Grammatiken und reguläre Grammatiken 51

2.7 Zusammenfassung 53

3 Kontextfreie Sprachen **55**

3.1 Ableitungsbäume 56

3.2 Die Chomsky-Normalform und das Wortproblem 59

3.3 Das Pumping Lemma und Ogden's Lemma 69

3.3.1 Abschlusseigenschaften kontextfreier Sprachen 73

3.4 Kellerautomaten 74

3.4.1 Deterministisch kontextfreie Sprachen 80

3.4.2 Abschlusseigenschaften deterministisch kontextfreier Sprachen 82

3.5 Entscheidungsprobleme 83

3.5.1 Das Postsche Korrespondenzproblem 84
3.5.2 Unentscheidbare Probleme für kontextfreie Sprachen 86
3.5.3 Das Äquivalenzproblem für Finite State Transducer 87
3.6 Zusammenfassung 88

4 XML und reguläre Baumgrammatiken* **91**
4.1 XML 91
4.2 DTDs 94
4.3 Reguläre Baumsprachen 96
4.3.1 Baumsprachen mit eindeutigen Typen 99
4.3.2 Reguläre Baumgrammatiken und Baumautomaten 101
4.4 XPath und XQuery 105

II Komplexitätsklassen **107**

5 Speicherplatz-Komplexität **109**
5.1 Eine Platzhierarchie 111
5.2 Sub-Logarithmischer Speicherplatz 112
5.3 Logarithmischer Speicherplatz 114
5.3.1 DL 114
5.3.2 NL und NL-Vollständigkeit 117
5.3.3 Der Satz von Savitch 119
5.3.4 Der Satz von Immerman und Szlepscenyi 121
5.4 PSPACE-Vollständigkeit 123
5.4.1 QBF: Quantifizierte Boolesche Formeln 124
5.4.2 Das Geographie-Spiel 127
5.4.3 NFA's und reguläre Ausdrücke 127
5.5 Komplexitätsklassen und die Chomsky Hierarchie 129
5.6 Probabilistische Turingmaschinen und Quantenrechner 134
5.7 Zusammenfassung 137

6 Parallelität **139**
6.1 Parallele Rechenzeit versus Speicherplatz 142
6.2 P-Vollständigkeit 144
6.2.1 Das Circuit Value Problem 145
6.2.2 Die Lineare Programmierung 148
6.2.3 Parallelisierung von Greedy-Algorithmen 149
6.3 Zusammenfassung 151

Kapitel 1

Einführung

1.1 Ziele der Vorlesung

Die Vorlesung „Theoretische Informatik 2“ beschließt den Grundstudiumszyklus der theoretischen Informatik.

- In der Veranstaltung „Diskrete Modellierung“ werden fundamentale diskrete Modellierungsmethoden zur Verfügung gestellt. Behandelt werden etwa Logiken wie die Aussagen- und Prädikatenlogik, Bäume und Graphen, endliche Automaten, Markovketten und kontextfreie Grammatiken.
- Die Veranstaltungen „Datenstrukturen“ und „Algorithmentheorie“ sind vor Allem der Beschreibung von Datenstrukturen, Standardalgorithmen und Entwurfsmethoden sowie der Effizienzmessung von Algorithmen gewidmet. Desweiteren wird mit den Konzepten der `NP`-Vollständigkeit und der Entscheidbarkeit die Möglichkeit gegeben, die Komplexität algorithmischer Probleme zu bestimmen. Die Vorlesungsinhalte bilden das Rüstzeug für den Entwurf effizienter Algorithmen und das Erkennen schwieriger algorithmischer Probleme.

Im ersten Teil dieser Vorlesung wird unser Hauptaugenmerk dem Compilerproblem, bzw. dem Wortproblem gelten, nämlich der Frage, ob ein vorgelegtes Programm einer höheren Programmiersprache syntaktisch korrekt ist, und wir werden Typ Checking für XML-Dokumente durchführen. Dazu werden wir bereits aus der „diskreten Modellierung“ bekannte Konzepte wie endliche Automaten und kontextfreie Grammatiken wieder aufgreifen und vertiefen sowie neue Konzepte wie unbeschränkte (konventionelle) Grammatiken und Baumgrammatiken untersuchen.

Im zweiten Teil fragen wir uns, wieviel Ressourcen benötigt werden, um das Wortproblem für wichtige Klassen formaler Sprachen –wie etwa reguläre, kontextfreie oder kontextsensitive Sprachen– zu lösen. Neben den schon aus der Theoretischen Informatik 1 bekannten Komplexitätsklassen `P` und `NP` spielen auch Komplexitätsklassen eine wichtige Rolle, die den notwendigen Speicherverbrauch messen: Wir werden auf die Komplexitätsklassen `DL` und `NL` wie auch auf die Klasse `PSPACE` geführt. Diese Klassen sind nicht nur für das Verständnis formaler Sprachen wichtig, sondern auch für das Verständnis nicht-klassischer Berechnungen wie randomisierte Berechnungen oder Quantenberechnungen. Als Nebenprodukt werden wir auch zum Beispiel sehen, dass das Entscheidungsproblem „Besitzt der ziehende Spieler einen Gewinnzug?“ für viele nicht-triviale Zweipersonenspiele zu den schwierigsten Problemen der Klasse `PSPACE` gehört: Man kann deshalb die Klasse `PSPACE` auch zur Klassifikation der Schwierigkeit von Spielen benutzen!

Welche Probleme erlauben ultra-schnelle parallele Algorithmen? Wir betrachten zur Beantwortung dieser Frage die Klasse NC aller parallelisierbaren Entscheidungsprobleme: Ein Entscheidungsproblem L gehört genau dann zu NC, wenn L rasant schnelle Algorithmen besitzt, die mit nicht zu vielen Prozessoren arbeiten. Wir fragen uns dann, welche Probleme in P nicht parallelisierbar sind und führen zur Beantwortung dieser Frage die LOGSPACE-Reduktion und den Begriff der P-Vollständigkeit ein. Wir stellen fest, dass P = NC genau dann gilt, wenn eine P-vollständige Sprache parallelisierbar ist: Die P-vollständigen Sprachen sind also die im Hinblick auf Parallelisierbarkeit schwierigsten Sprachen in P. Schließlich werden wir auf einen sehr engen Zusammenhang zwischen Parallelisierbarkeit und Speicherplatz-Komplexität geführt, wenn wir die Anzahl der Prozessoren außer Acht lassen.

1.2 Begriffe und Notationen

Wir führen die für die Vorlesung wichtigsten Begriffe und Notationen ein:

Definition 1.1

(a) $\mathbb{N} = \{0, 1, 2, \ldots\}$ *bezeichnet die Menge der natürlichen Zahlen,* $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$ *ist die Menge der ganzen Zahlen.*

(b) $\mathbb{Q} = \{r/s \,|\, r \in \mathbb{Z}, s \in \mathbb{N}, s \neq 0\}$ *bezeichnet die Menge der rationalen Zahlen.*

(c) $\mathbb{R}$ *ist die Menge der reellen Zahlen und*

(d) $\mathbb{C}$ *ist die Menge der komplexen Zahlen.*

Als nächstes führen wir den Begriff einer formalen Sprache ein:

Definition 1.2 (Alphabete, Worte und Sprachen)

(a) Ein Alphabet Σ *ist eine endliche, nicht-leere Menge. Die Elemente von* Σ *werden Buchstaben genannt.*

(b) $\Sigma^n = \{(a_1, \ldots, a_n) \,|\, a_i \in \Sigma\}$ *ist die Menge aller Worte der Länge* n *über* Σ*. Wir werden im folgenden* $a_1 \cdots a_n$ *statt* $(a_1, \ldots, a_n)$ *schreiben.*

(c) Wir definieren $\Sigma^0 = \{\varepsilon\}$ *als die Menge, die nur das* leeren Wort ε *besitzt.*

(d) $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$ *ist die Menge aller Worte über dem Alphabet* Σ*.*

(e) $\Sigma^+ = \bigcup_{n=1}^{\infty} \Sigma^n$ *ist die Menge aller nicht-leeren Worte über* Σ*.*

(f) Für $w \in \Sigma^*$ *bezeichnet* $|w|$ *die Länge von* w*, also die Anzahl der Buchstaben von* w*.*

(g) Eine (formale) Sprache L *(über* Σ*) ist eine Teilmenge von* Σ^**.*

Als nächstes besprechen wir die wichtigsten Begriffe im Umgang mit formalen Sprachen.

Definition 1.3 *Sei* Σ *ein Alphabet,* u *und* v *seien Worte über* Σ*.*

(a) *Es sei* $u = u_1 \cdots u_n$ *und* $v = v_1 \cdots v_m$*. Dann bezeichnet*

$$u \cdot v = u_1 \cdots u_n \cdot v_1 \cdots v_m$$

die *Konkatenation* *von* u *und* v*. (Beachte, dass* $u \cdot \varepsilon = \varepsilon \cdot u = u$*). Wir schreiben auch* uv *statt* $u \cdot v$*.*

(b) u *heißt genau dann ein* *Teilwort* *von* v *wenn es Worte* u_1 *und* u_2 *über* Σ *gibt mit*

$$v = u_1\, u\, u_2.$$

(c) u *heißt genau dann* *Präfix* *von* v *wenn es ein Wort* u_1 *über* Σ *mit*$v = u\, u_1$ *gibt.*

u *heißt genau dann ein* *Suffix,* *wenn es ein Wort* u_1 *über* Σ *mit* $v = u_1\, u$ *gibt.*

(d) *Für Sprachen* L_1, L_2 *über* Σ *bezeichnet*

$$L_1 \circ L_2 = \{uv \mid u \in L_1, v \in L_2\}$$

die Konkatenation von L_1 *und* L_2*.*

(e) *Für eine Sprache* L *über* Σ *ist*

$$\begin{aligned} L^n &= \{u_1 \cdots u_n \mid u_1, \ldots, u_n \text{ sind Worte der Sprache } L\} \\ L^* &= \bigcup_{n=0}^{\infty} L^n \quad (\text{mit } L^0 = \{\varepsilon\}) \\ L^+ &= \bigcup_{n=1}^{\infty} L^n. \end{aligned}$$

L^* *heißt die* *Kleene Hülle* *von* L*.*

Beispiel 1.1 (a) $\{A, B, C, D, E, F, G, H\} \circ \{1, 2, 3, 4, 5, 6, 7, 8\}$ ist die Menge aller Felder eines Schachbretts.

(b) $\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\} \circ \{7, 8, 9, 10, \mathit{Bube}, \mathit{Dame}, \mathit{König}, \mathit{Ass}\}$ ist die Menge der Karten eines Skatblatts.

(c) $\{1\} \circ \{0, 1\}^*$ ist die Menge der Binärdarstellungen der natürlichen Zahlen größer Null.

(d) $\bigcup_{i=1}^{4} \{\cdot, -\}^i$ ist die Menge der Morsezeichen für die 26 Buchstaben, ä, ö, ü und ch.

(e) $(\{2\} \circ \{0, 1, 2, 3\} \,\cup\, \{\varepsilon, 1\} \circ \{0, 1, \ldots, 9\}) \circ \{:\} \circ \{0, 1, \ldots, 5\} \circ \{0, 1, \ldots, 9\}$ ist die Menge der Uhrzeiten eines Tages.

(f) $(\{1\} \circ \{0, 1, 2\} \,\cup\, \{1, \ldots, 9\}) \circ \{:\} \circ \{0, 1, \ldots, 5\} \circ \{0, 1, \ldots, 9\} \circ \{am, pm\}$ ist die amerikanische Entsprechung.

Als nächstes spielen wir ein wenig mit der Konkatenation und mit dem Sternoperator, also mit der Kleene Hülle.

Beispiel 1.2 Wir behaupten, dass das Wort *abcd* ein Element der Sprache

$$(\{a\}^* \circ \{b\}^* \circ \{d\}^* \circ \{c\}^*)^* = L$$

ist. Dazu betrachten wir zuerst die Sprache

$$K = \{a\}^* \circ \{b\}^* \circ \{d\}^* \circ \{c\}^*.$$

Dann ist

$$K = \{a^n\ b^m\ d^k\ c^l \mid n, m, k, l \in \mathbb{N}\}$$

und das Wort *abcd* gehört nicht zu K. Beachte aber, dass

$$L = K^*$$

und

$$abcd \in K^2$$

(denn $abc \in K$ und $d \in K$). Somit ist also $abcd \in L$. Die Sprache L können wir aber noch einfacher charakterisieren. Wir behaupten nämlich, dass

$$L = \{a, b, c, d\}^*.$$

Wir zeigen zuerst $L \subseteq \{a, b, c, d\}^*$. Dies ist offensichtlich, da L nur aus Worten über dem Alphabet $\{a, b, c, d\}$ besteht, und da $\{a, b, c, d\}^*$ die Menge aller Worte über $\{a, b, c, d\}$ ist. Zuletzt ist die umgekehrte Inklusion

$$\{a, b, c, d\}^* \subseteq L$$

nachzuweisen. Sei w ein beliebiges Wort über dem Alphabet $\{a, b, c, d\}$. Wir müssen zeigen, dass w auch ein Element von L ist. Wenn $w = w_1 \cdots w_n$ mit $w_1, \ldots, w_n \in \{a, b, c, d\}$, dann ist natürlich $w_1, \ldots, w_n \in K$: jeder Buchstabe von w gehört zur Sprache K. Dann muß aber

$$w_1 \cdots w_n \in K^n$$

gelten und somit ist $w = w_1 \cdots w_n \in L$. □

Zur Definition formaler Sprachen wird oft das Schema der *rekursiven Definition* gewählt.

Beispiel 1.3 Die Sprache L sei die kleinste Sprache über dem Alphabet $\Sigma = \{0, 1\}$ mit den folgenden Eigenschaften

(a) $\varepsilon \in L$

(b) wenn $u \in L$, dann $0u \in L$ und $u1 \in L$.

Wie sieht L aus? Wir geben eine einfache, nicht rekursive Beschreibung, nämlich

$$L = \{0\}^*\ \{1\}^*.$$

Beweis: Erst zeigen wir $\{0\}^*\ \{1\}^* \subseteq L$. Sei $w \in \{0\}^*\ \{1\}^*$ und demgemäß $w = 0^n\ 1^m$. Wir wissen, dass $\varepsilon \in L$ und deshalb ist

$$0 \cdot \varepsilon, 0 \cdot 0 \cdot \varepsilon, \ldots, 0^n \cdot \varepsilon \in L.$$

Mit $0^n\varepsilon = 0^n$ ist aber auch

$$0^n \cdot 1, 0^n \cdot 1 \cdot 1, \cdots, 0^n\ 1^m \in L,$$

und deshalb ist $w \in L$.

Es bleibt zu zeigen, dass $L \subseteq \{0\}^*\ \{1\}^*$. Dazu genügt es zu bemerken, dass die Sprache $\{0\}^*\ \{1\}^*$ die Eigenschaften (1) und (2) in der Definition von L erfüllt. (Warum ist dies ausreichend? L ist die *kleinste* Sprache mit den Eigenschaften (1) und (2)!). □

Ein wesentliches Mittel zur Beschreibung von Sprachen sind reguläre Ausdrücke. Wir definieren sie wie folgt.

Definition 1.4 *Die Menge der regulären Ausdrücke über einem endlichen Alphabet Σ wird rekursiv durch die folgenden Regeln definiert.*

(a) Die Ausdrücke $\emptyset$, ε und a für $a \in \Sigma$ sind regulär. Die Ausdrücke stellen die leere Sprache, die Sprache des leeren Wortes und die Sprache des einbuchstabigen Wortes a dar.

(b) Sind A_1 und A_2 reguläre Ausdrücke, dann auch $(A_1) + (A_2)$, $(A_1) \circ (A_2)$ und $(A_1)^*$, wobei $+$ die Vereinigung, $\circ$ die Konkatenation und $*$ den Kleeneschen Abschluß darstellen.

(c) Alle regulären Ausdrücke lassen sich durch endliche Anwendung der Regeln (a) und (b) erzeugen.

Aufgabe 1
Die Sprache L (über dem Alphabet $\Sigma = \{a, b\}$) sei die kleinste Wortmenge mit den Eigenschaften „der Buchstabe a gehört zu L“ und „wenn das Wort x zu L gehört, so gehören auch die Worte bx, ax und xb zu L“.

Finde einen regulären Ausdruck für die Sprache L und **beweise**, dass deine Beschreibung zu der obigen rekursiven Definition äquivalent ist.

Aufgabe 2
In **welchen Beziehungen** stehen die drei Sprachen L_1, L_2 und L_3 zueinander? Es soll paarweise entschieden werden, ob Gleichheit oder Teilmengenbeziehung besteht.

$$
\begin{aligned}
L_1 &= \left(\{a\}^* \circ \{b\}^* \circ \{c\}^+\right)^* \\
L_2 &= \left(\{a, b\}^* \circ \{c\}^+\right)^* \\
L_3 &= \left((\{a\}^* \circ \{c\}^+) \cup (\{b\}^* \circ \{c\}^+)\right)^*
\end{aligned}
$$

Aufgabe 3
Finde je einen regulären Ausdruck für die beiden folgenden Sprachen über $\Sigma = \{a, b, c\}$.

(a) $L_1 := \{w \in \Sigma^* | w$ enthält das Teilwort $abc\}$

(b) $L_2 := \{w \in \Sigma^* | w$ enthält das Teilwort abc **nicht**$\}$

1.3 Das Wortproblem

Das Wortproblem für eine Sprache L ist denkbar einfach zu beschreiben:

> Es ist zu entscheiden, ob ein gegebenes Wort w in L liegt.

Die Lösung des Wortproblems für L besteht in der Angabe eines möglichst effizienten Algorithmus. Dass das Wortproblem im Schwierigkeitsgrad stark schwankt, ist schnell überlegt. Das Wortproblem für die Sprache

$$L = \{Frühling, Sommer, Herbst, Winter\}$$

ist leicht, da zu jedem Wort w schnell festgestellt werden kann, ob es eines der vier enthaltenen ist.

Nehmen wir als anderes Beispiel die Sprache L, die über dem ASCII Zeichensatz als Alphabet Σ definiert sei und als Worte alle syntaktisch korrekten C-Programme enthalten soll, die auf jede Eingabe hin anhalten. Wie ist es mit diesem Kandidaten w:

```
main(int n)
int i;
{   i=n;
    while(NOT(prim(i)&&prim(i+2)))
       i++;
}

boolean prim(int n)
int i;
{   for(i=2;i<n;i++)
       if(n % i == 0) return(false);
    return(true);
}
```

Was tut dieses Programm? Es beginnt bei der Zahl n aufsteigend nach Primzahlzwillingen zu suchen und hört nur auf, wenn es fündig wird. Die Frage, ob dieses Programm w in unsere Sprache L gehört, ist also gleichbedeutend mit der Frage, ob es unendlich viele Primzahlzwillinge gibt. Das Programm hält nämlich genau dann für Eingabe n, wenn es zu n ein Paar von größeren Primzahlzwillingen gibt. Ob das so ist, ist ein offenes Forschungsproblem.

Wir wollen in der Vorlesung natürlich nicht für jede betrachtete Sprache L die Schwierigkeit des Wortproblems individuell analysieren. Wir werden uns stattdessen auf Sprachklassen der Chomsky-Hierarchie beschränken. In ersten Teil der Vorlesung wird uns das Wortproblem als so genanntes *Compilerproblem* begegnen. Die Frage ist dabei, ob eine vorgelegte Zeichenkette ein syntaktisch korrektes Programm darstellt.

Im nächsten Kapitel führen wir beispielsweise die *regulären Sprachen* als die formalen Sprachen ein, die durch einen regulären Ausdruck beschrieben werden. Wir werden zeigen, dass *endliche Automaten* das Wortproblem für alle regulären Sprachen lösen können. Allerdings sind regulären Sprachen enge Grenzen gesetzt. Nehmen wir etwa als Beispiel die Sprache L der legalen Klammerausdrücke, die wir wie folgt rekursiv definieren.

(a) $\varepsilon \in L$

(b) $u, v \in L \rightarrow u \circ v \in L$

(c) $w \in L \rightarrow \{[\} \circ w \circ \{]\} \in L$

(d) L ist die kleinste Sprache, die diese Bedingungen erfüllt.

Zu dieser sehr simplen Sprache wird es auch nach längerem Bemühen nicht gelingen, einen regulären Ausdruck anzugeben. Mit unseren rekursiven Definitionen, die sich selbst referenzieren dürfen, hätten wir kein Problem:

$$KLAMMER = \\ \{\varepsilon\} \cup (KLAMMER \circ KLAMMER) \cup (\{[\} \circ KLAMMER \circ \{]\}).$$

Wir haben es hier mit einer *kontextfreien Ableitungsvorschrift* zu tun. Die entsprechende Sprachklasse der kontextfreien Sprachen werden wir im Kapitel 3 untersuchen.

Programmiersprachen werden in der Tat durch solche formalen Definitionen beschrieben. Man denke an die erweiterte Backus Naur Form oder an Syntaxdiagramme. Eine IF Anweisung in C

ist beispielsweise definiert als:

$$\begin{aligned} IF &= \{\texttt{if}\} \circ \{(\} \circ EXPRESSION \circ \{)\} \circ STATEMENT \circ \\ &\quad (\{\varepsilon\} \cup (\{\texttt{else}\} \circ STATEMENT)) . \end{aligned}$$

Eine IF-Anweisung ist also ein `if` gefolgt von einem geklammerten Ausdruck. Eine Anweisung folgt. Daran kann, muss aber nicht, sich ein Paar von Schlüsselwort `else` und einer Anweisung anschließen.

Für einen Ausdruck, bietet C eine Fülle von Formen

$$\begin{aligned} & EXPRESSION = CONSTANT \cup TERM \\ \cup\;\; & EXPRESSION \circ BINOP \circ EXPRESSION \\ \cup\;\; & TERM \circ ASGNOP \circ EXPRESSION \\ \cup\;\; & EXPRESSION \circ \{?\} \circ EXPRESSION \circ \{:\} \circ EXPRESSION \\ \cup\;\; & \{\&, ++, --\} \circ TERM \\ \cup\;\; & TERM \circ \{++, --\} \\ \cup\;\; & \{-, !, \sim\} \circ EXPRESSION \\ \cup\;\; & \{(\} \circ TYPENAME \circ \{)\} \circ EXPRESSION \\ \cup\;\; & \{(\} \circ EXPRESSION \circ \{)\}. \end{aligned}$$

Dies läßt sich bis zu den Zeichen des Zeichensatzes herunterbrechen. $BINOP$, also ein binary Operator, ist wie folgt definiert,

$$\begin{aligned} BINOP &= BINOP_A \cup BINOP_B \\ BINOP_A &= \{*, /, \%, +, -, \gg, \ll, >, <, <=, >=, ==, !=, \&, \hat{}, |\} \\ BINOP_B &= \{\&\&, ||, , \}. \end{aligned}$$

Es lassen sich allerdings nicht alle Anforderungen, die eine Programmiersprache an einen Quellcode stellt, auf diese Weise beschreiben. Wir können zwar erzwingen, dass auf jede öffnende Klammer eine schließende folgt, dass kein `else` ohne vorheriges `if` auftritt, aber wir können beispielsweise nicht artikulieren, dass eine Variable deklariert sein muss, bevor man sie verwendet.

Wir werden sehen, dass bei kontextfreien Sprachen schärfere Geschütze aufgefahren werden müssen, um das Wortproblem in den Griff zu bekommen. Der Algorithmus von Cocke Younger und Kasami wird das Wortproblem für kontextfreie Sprachen in Zeit $\Theta(n^3)$ für Eingaben der Länge n lösen. Damit sind wir zwar im polynomiellen Bereich, aber für praktische Anwendungen ist das zu langsam. Die Einschränkung auf deterministisch kontextfreie Sprachen wird hier helfen, den optimalen Tradeoff zwischen der Ausdrucksstärke der Sprachen und der Effizienz der Compilierung zu finden.

Für beide Klassen, reguläre und kontextfreie Sprachen, werden wir Techniken herleiten, um formal nachzuweisen, dass eine gegebene Sprache nicht regulär bzw. nicht kontextfrei ist.

1.4 Komplexitätsklassen

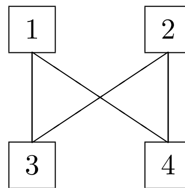
Das Konzept des Wortproblems hat extrem viele Anwendungen, alle algorithmischen Entscheidungsprobleme lassen sich zum Beipiel als Wortprobleme auffassen.

Beispiel 1.4 Einen ungerichteten Graphen $G = (V, E)$ können wir durch seine Adjazenzmatrix repräsentieren. Die Adjazenzmatrix wiederum lässt sich durch zeilenweises Hintereinandersetzen als ein binärer String auffassen. Wir können damit das Problem der Hamiltonschen Kreise formulieren:

$$L = \{w \in \{0,1\}^* |\ w \text{ beschreibt einen Graphen mit Hamiltonischen Kreis}\}$$

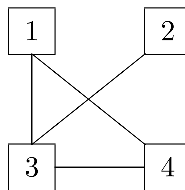
Das Wortproblem wird so zum Entscheidungsproblem „Hat ein durch seine Kodierung gegebener Graph einen Hamiltonischen Kreis, also einen Kreis, der jeden Knoten genau einmal durchläuft?“

(a) Das Wort $w_1 = 0011001111001100$ gehört zur Sprache L, da das Wort den Graphen



beschreibt, der einen Hamiltonischen Kreis besitzt.

(b) Das Wort $w_2 = 0011001011011010$ gehört **nicht** zur Sprache L, da das Wort den Graphen



beschreibt, der keinen Hamiltonischen Kreis besitzt.

(c) Das Wort $w_3 = 0010101101$ gehört **nicht** zur Sprache L, da das Wort keinen Graphen beschreibt: $|w_3|$ ist keine Quadratzahl.

(d) Das Wort $w_4 = 0001000000000000$ gehört **nicht** zur Sprache L, da der String keiner Adjazenzmatrix entspricht.

Beispiel 1.5 Wir können über dem Alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \diamond\}$ Folgen natürlicher Zahlen, durch $\diamond$ getrennt, kodieren. So können wir die Sprache

$$L = \left\{ w \in \Sigma^* \middle| \begin{array}{l} w \text{ beschreibt eine Zahlenfolge, so dass eine} \\ \text{aufsteigende Teilfolge existiert, die mindestens} \\ \text{halb so lang ist wie die Gesamtfolge} \end{array} \right\}$$

definieren.

(a) Das Wort $w_1 = 9 \diamond 15 \diamond 21 \diamond 4 \diamond 11$ gehört zur Sprache L, da die Zahlenfolge 9,15,21,4,11 repräsentiert wird, und die Teilfolge 9,15,21 mindestens halb so lang ist wie die Gesamtfolge.

(b) Das Wort $w_2 = 17 \diamond 10 \diamond 21 \diamond 5 \diamond 19$ gehört **nicht** zur Sprache L, da die repräsentierte Zahlenfolge 17, 10, 21, 5, 19 keine aufsteigende Teilfolge mit Länge größer als 2 besitzt.

(c) Das Wort $w_3 = 005 \diamond \diamond \diamond 33$ gehört **nicht** zur Sprache, da es keine Zahlenfolge kodiert.

Beispiel 1.6 Wie wir es oben bereits getan haben, können wir auch den kompletten ASCII Zeichensatz als Σ wählen. Dann können wir definieren:

$$L = \{w \in \Sigma^* | w \text{ ist eine C-Funktion, die zu gegebenem } n \text{ den Wert } n^2 \text{ ausgibt}\}$$

Das Wort

$$w_1 = \texttt{int quad(int n)\{return(n * n);\}}$$

gehört zu L, da die Funktion in der Tat n^2 berechnet. Das Wort

$$w_2 = \texttt{quad(int n)\{printf("hello");\}}$$

gehört **nicht** zur Sprache, da die repräsentierte Funktion nicht n^2 berechnet. Schließlich gehört das Wort

$$w_3 = \texttt{Lieber Compiler, bitte berechne mir n zum Quadrat, Danke!}$$

nicht zur Sprache L, da dies beim derzeitigen Stand der Compilertechnik kein C-Programm darstellt.

Die Komplexitätstheorie beschäftigt sich mit der Frage, welche Ressourcen ein Algorithmus benötigt, der ein vorgelegtes Entscheidungsproblem lösen soll.

- Das Problem des Hamiltonischen Kreises (siehe Beispiel 1.4) gehört zu den NP-vollständigen Problemen. Das Finden einer effizienten Lösung, das heisst eines Algorithmus mit polynomieller Laufzeit wäre eine Sensation.
- Das Zahlenfolgenproblem (siehe Beispiel 1.5) kann mit dynamischer Programmierung in polynomieller Zeit gelöst werden. Das Zahlenfolgenproblem liegt damit in der Klasse P, der in polynomieller Zeit lösbaren Entscheidungsprobleme.
- Die Frage, ob ein gegebenes Programm eine bestimmte Aufgabe erfüllt (siehe Beispiel 1.6), ist **nicht** entscheidbar, selbst wenn keine Zeitbeschränkung gegeben ist. Diese absolute Grenze algorithmischer Berechnungskraft haben wir in der Theoretischen Informatik 1 kennengelernt.

Das Wortproblem hat auch Implikationen über Entscheidungsprobleme, wie die hier dargestellten, hinaus. Das aus der Theoretischen Informatik 1 bekannte Rucksackproblem ist zum Beispiel ein Optimierungsproblem. Die Frage ist, wieviel *Wert* kann transportiert werden, wenn n Objekte mit den Einzelwerten $w_1, w_2, \ldots, w_n$ und den Gewichten $g_1, \ldots, g_n$ zur Verfügung stehen, die ganz oder gar nicht mitgenommen werden können, und die Kapazität C nicht überschritten werden darf. In dieser Formulierung ist die Antwort offenbar nicht Ja oder Nein, sondern eine Zahl, nämlich der Wert einer optimalen Bepackung.

Zur Formulierung als Entscheidungsproblem führen wir noch einen Sollwert S ein. Die Frage lautet nun, ob bei gegebenen Objekten und gegebener Kapazität der Wert S erreicht oder übertroffen werden kann. Nun haben wir ein Entscheidungsproblem, dessen Instanzen wir in eine Zeichenkette kodieren können. Etwa:

$$w_1 \diamond g_1 \diamond w_2 \diamond g_2 \diamond \ldots \diamond w_n \diamond g_n \diamond \diamond C \diamond S$$

Nehmen wir nun an, wir könnten dieses Wortproblem in Zeit $f(n)$ lösen, so können wir wie folgt vorgehen, um das Ausgangsproblem zu lösen.

1. Setze l auf 0 und r auf die Summe aller Werte.
2. Solange $l < r$
   a) Setze $m = \lceil (l+r)/2 \rceil$.
   b) Löse die Entscheidungsversion mit $S = m$.
   c) Falls Antwort positiv, setze $l = m$ anderenfalls $r = m - 1$.
3. Gib l als Lösung aus.

Eine simple Binärsuche löst also das Optimierungsproblem mit Hilfe des Entscheidungsproblems. Wie lange dauert die Suche? Wenn W die Summe aller Werte ist, dann wird die Schleife höchstens $O(\log_2(W))$ mal durchlaufen. Der Aufwand innerhalb einer Iteration ist im wesentlichen der Aufwand des Entscheidungsproblems und wir erhalten

$$O(f(n) \cdot \log_2 W)$$

als Gesamtlaufzeit, denn wir haben angenommen, dass sich das Wortproblem in Zeit $f(n)$ lösen lässt. Man beachte, dass $\log_2 W$ aber nicht größer als die Länge der Binärdarstellungen der Werte und somit ist $\log_2 W$ nicht größer als die Eingabelänge n.

Interessieren wir uns also nur dafür, ob ein Optimierungsproblem in polynomieller Zeit gelöst werden kann oder nicht, so wird uns die *Entscheidungsversion* bereits die richtige Antwort liefern. In den Übungen werden wir sehen, dass auch das Problem des Auffindens der optimalen Bepackung selbst – im Gegensatz zum Wert – in derselben komplexitätstheoretischen Preisklasse anzusiedeln ist.

Wir werden uns im zweiten Teil der Vorlesung mit den wichtigsten Komplexitätsklassen befassen, die nicht die Laufzeit, sondern den Speicherplatzverbrauch in der Lösung von Entscheidungsproblemen messen. Diese Klassen haben vielfältige Anwendungen.

1. Randomisierte Algorithmen und sogar Quantenalgorithmen, die „nur" polynomielle Zeit benötigen, können durch konventionelle deterministische Algorithmen mit polynomiellem Speicherplatzverbrauch simuliert werden. Dieses Ergebnis hilft in der Einschätzung der Berechnungskraft von Randomisierung wie auch von Quantenberechnungen.
2. Hat der ziehende Spieler einen Gewinnzug (in einem bestimmten zwei-Personenspiel)? Für viele interessante Spiele führt diese Frage auf Berechnungen, die höchstwahrscheinlich nicht in polynomieller Laufzeit, wohl aber mit polynomiellen Speicherplatz gelingen.
3. Zurück zu den formalen Sprachen: Selbst einfach klingende Entscheidungsprobleme wie etwa

   Sind zwei nichtdeterministische endliche Automaten äquivalent?

   gehören bereits zu den schwierigsten Problemen, die mit polynomiellem Speicherplatz lösbar sind. Wir werden auch eine neue Klasse formaler Sprachen, die Klasse der kontextsensitiven Sprachen kennenlernen: Diese Sprachklasse wird sich mit Hilfe der Speicherplatzkomplexität exakt charakterisieren lassen.

Aufgabe 4
Überflüssige Programmteile: Wir fragen, ob zum Compilierungszeitpunkt algorithmisch festgestellt werden kann, ob ein gegebenes C-Programm eine Anweisung enthält, die für keine Eingabe zur Ausführung kommt. In der Regel würde die Existenz einer solchen Anweisung auf einen Programmierfehler hindeuten.

Beurteile die Komplexität dieses Problems und **belege** Deine Einschätzung möglichst schlüssig.

1.5 Literatur

Die folgenden Textbücher sind besonders empfehlenswert.

- M. Sipser, Introduction to the Theory of Computation, Paperback 3rd edition, Cengage Learning, 2012.
- I. Wegener, Theoretische Informatik, B.G. Teubner 1993.
- I. Wegener, Kompendium Theoretische Informatik - eine Ideensammlung, B.G. Teubner, 1996.
- U. Schöning, Theoretische Informatik - kurzgefasst, Spektrum 1997.
- J.E. Hopcroft, J.D. Ullman, R. Motwani, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 2001.

Teil I

Formale Sprachen

Kapitel 2

Endliche Automaten und reguläre Sprachen

Unser erstes Ziel ist die Definition endlicher Automaten, eines Maschinenmodells, das seine Eingabe von links nach rechs liest und ohne Speicher auskommen muss. Die Beschreibung eines endlichen Automaten besteht aus den folgenden Komponenten:

- dem Eingabealphabet Σ,
- der endlichen Menge Q der Zustände,
- dem Anfangszustand $q_0 \in Q$,
- der Menge F der akzeptierenden Zustände
- sowie dem Programm (bzw. der Zustandsüberführungsfunktion) $\delta\colon Q \times \Sigma \to Q$.

Wie rechnet ein endlicher Automat? Beginnend mit einem Startzustand q_0 wird die Eingabe Buchstabe für Buchstabe eingelesen und bewirkt Zustandsänderungen. Die Eingabe wird akzeptiert, wenn der zuletzt erhaltene Zustand akzeptierend ist.

Definition 2.1 *Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein endlicher Automat.*

(a) A liest ein Wort $w \in \Sigma^*$ von links nach rechts und wechselt seine Zustände gemäß dem Programm δ. Formal:

Wir setzen δ auf Worte in Σ^* fort und definieren

$$\delta\colon Q \times \Sigma^* \to Q$$

durch

$$\begin{aligned} \delta(q, \varepsilon) &= q, \\ \delta(q, w_1 \cdots w_{n+1}) &= \delta\Big(\delta(q, w_1 \cdots w_n), w_{n+1}\Big) \end{aligned}$$

Beachte, dass $\delta(q, w_1 \cdots w_n)$ den Zustand bezeichnet, den A nach Lesen der Eingabe $w_1 \cdots w_n$ erreicht, wenn im Zustand q begonnen wird.

(b) A akzeptiert $w \in \Sigma^*$ genau dann, wenn $\delta(q_0, w) \in F$.

(c) *$L(A) = \{w \in \Sigma^* \mid A$ akzeptiert $w\}$ ist die von A akzeptierte (oder erkannte) Sprache.*

(d) *Eine Sprache $L \subseteq \Sigma^*$ heißt regulär, wenn es einen endlichen Automaten A mit $L = L(A)$ gibt.*

Aufgabe 5
Zeige, dass eine Sprache L genau dann regulär ist, wenn L durch eine Turingmaschine erkannt wird, die keine Schreiboperationen ausführt.

Das Programm eines endlichen Automaten läßt sich am besten durch ein *Zustandsdiagramm* veranschaulichen, das aus Knoten und beschrifteten Kanten besteht. Die Knoten entsprechen den Zuständen; für jeden Befehl der Form $\delta(q, a) = q'$ wird eine mit a beschriftete Kante eingesetzt, die *vom* Knoten q *zum* Knoten q' verläuft.

Den Anfangszustand markieren wir durch einen Pfeil, akzeptierende Zustände werden doppelt umrandet.

Die Berechnung von A auf Eingabe $w = w_1 \cdots w_n$ beschreibt somit einen Weg im Zustandsgraphen, der

- im Knoten q_0 beginnt und
- jeweils die mit $w_1, w_2, \ldots, w_n$ beschrifteten Kanten wählt.

Beispiel 2.1 Wir behaupten, dass die Sprache

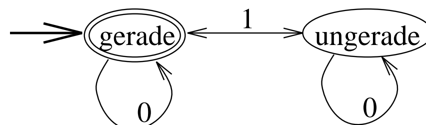
$$PARIT\ddot{A}T = \Big\{ w \in \{0,1\}^* \mid w \text{ hat gerade viele Einsen} \Big\}$$

regulär ist. $PARIT\ddot{A}T$ wird von einem endlichen Automaten mit

- Zustandsmenge $Q = \{\text{gerade}, \text{ungerade}\}$,
- Anfangszustand gerade,
- $F = \{\text{gerade}\}$ und
- dem Programm δ akzeptiert, wobei

$$\begin{array}{ll} \delta(\text{gerade}, 0) = \text{gerade}, & \delta(\text{gerade}, 1) = \text{ungerade}, \\ \delta(\text{ungerade}, 0) = \text{ungerade}, & \delta(\text{ungerade}, 1) = \text{gerade}. \end{array}$$

Das Zustandsdiagramm hat die Form



Beispiel 2.2 Üben wir uns in der Modellierung und betrachten das folgende Problem. Die Kindersicherung eines Fernsehers funktioniert so, dass über die Fernbedienung ein dreistelliger Code korrekt eingegeben werden muss, damit der Fernseher eingeschaltet werden kann. Dafür sind die Tasten von 0 bis 9 relevant. Es gibt eine `BACK` Taste, die es uns erlaubt, die letzteingegebene Zahl zurückzunehmen, außerdem gibt es die Taste `CODE`. Sie muss vor der Eingabe des Codes

gedrückt werden. Wird sie während der Codeeingabe noch mal gedrückt, so wird die Eingabe neu begonnen.

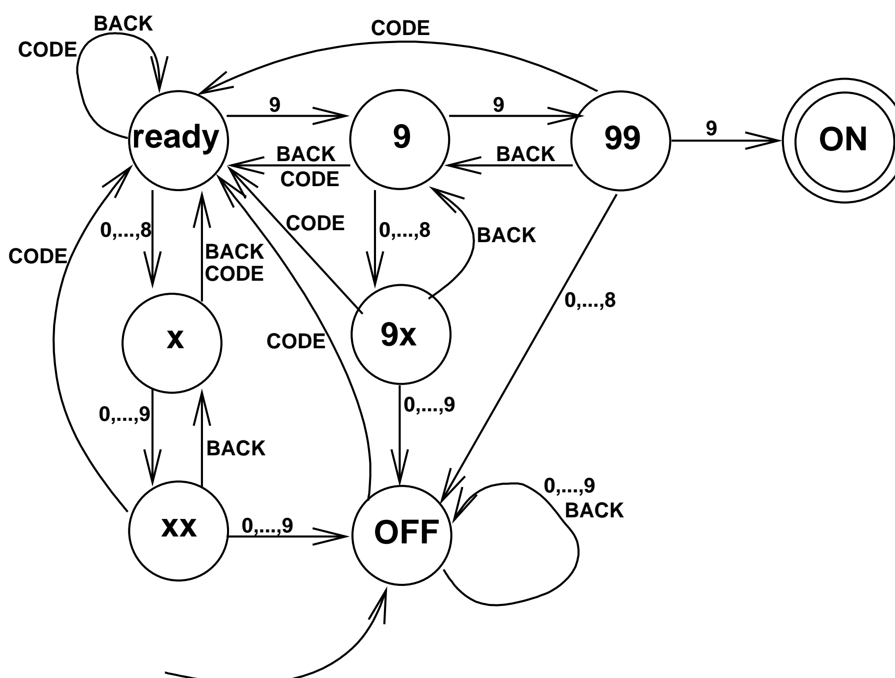
Ein Automat bietet sich für die Modellierung an, da die Eingabe, die Folge der Tastenbetätigungen, ohnehin nur von *links nach rechts* (in der zeitlichen Abfolge) zur Verfügung steht. Wir geben zunächst die Komponenten des Automaten an.

- Das Eingabealphabet sind die relevanten Tasten

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, BACK, CODE\}.$$

- Die Zustände müssen widerspiegeln, wie weit die Eingabe bereits fortgeschritten ist. Wie weit ist der Code richtig eingegeben worden? Folgen nach einem richtigen Anfang vielleicht falsche Zeichen, die mit `BACK` wieder zurückgenommen werden könnten? All diese Fragen müssen in den Zuständen beantwortet sein.
- Der Anfangszustand q_0 entspricht dem Zustand, in dem noch nichts eingegeben wurde.
- Die Menge der akzeptierenden Zustände hat nur ein Element: den Zustand, der nach erfolgreicher Codeeingabe angenommen wird.
- Die Übergangsfunktion δ wird die Zustände sinnvoll verbinden.

Ist der Code zum Entsperren des Fernsehers etwa 999, so erhalten wir den folgenden Automaten.



Bei der Erstellung dieses Automaten haben wir gesunden Menschenverstand eingesetzt. Wir haben erkannt, dass es nicht nötig ist zu unterscheiden, *welche* falsche Zahl getippt wurde. Auch haben wir ausgenutzt, dass es egal ist, ob man die zweite Zahl richtig eintippt, nachdem die erste falsch war. Sehr wohl spielt es aber eine Rolle, ob einer falschen zweiten Zahl eine richtige erste vorausging. Ist dieser Automat aber kleinstmöglich oder könnte man die Anforderungen kompakter, das heißt mit weniger Zuständen, formulieren?

Warum betrachten wir endliche Automaten? Endliche Automaten erlauben eine Beschreibung von Handlungsabläufen:

> Wie ändert sich ein Systemzustand in Abhängigkeit von veränderten Umgebungsbedingungen?

Dementsprechend ist das Einsatzgebiet endlicher Automaten vielfältig: Endliche Automaten finden Verwendung

- in der Entwicklung digitaler Schaltungen,
- in der Softwaretechnik (zum Beispiel in der Modellierung des Applikationsverhaltens),
- im Algorithmenentwurf für String Probleme
- oder in der Abstraktion tatsächlicher Automaten (wie Bank- und Getränkeautomaten)

Aufgabe 6
Ein Mann (M) steht mit einem Wolf (W), einer Ziege (Z) und einem Kohlkopf (K) am linken Ufer eines Flusses, den er überqueren will. Er hat ein Bot, das groß genug ist, ihn und ein weiteres Objekt zu transportieren, so dass er immer nur einen der drei mit sich hinübernehmen kann. Falls der Mann allerdings den Wolf und die Ziege unbewacht an einem der Ufer zurückläßt, so wird der Wolf sicherlich die Ziege fressen. Genauso wird, wenn die Ziege und der Kohlkopf unbewacht zurückbleiben, die Ziege den Kohlkopf fressen.

Eine Überführung ist ein Wort w über $\Sigma = \{M, W, Z, K\}$, wobei $w_i \in \{W, Z, K\}$, wenn der Mann mit Objekt w_i zum Zeitpunkt i das Boot benutzt und $w_i = M$, wenn der Mann das Bot alleine benutzt. Eine Überführung $w = w_1 \cdots w_n$ ist *erfolgreich*, falls nach der letzten Aktion w_n der Mann und alle drei Objekte das rechte Ufer erreicht haben.

Gebe einen möglichst kleinen Automaten **an**, der *alle* erfolgreichen Überführungen beschreibt.

Wir werden später sehen, dass die folgenden algorithmischen Probleme effiziente Lösungen besitzen:

- Das Äquivalenzproblem: Gegeben seien endliche Automaten A und B. Sind A und B äquivalent, d.h. gilt $L(A) = L(B)$?
- Das Minimierungsproblem: Gegeben sei ein endlicher Automat A. Bestimme einen äquivalenten endlichen Automaten B mit der kleinsten Anzahl von Zuständen.

Als Konsequenz des Minimierungsproblem kann man damit automatisch kleinste endliche Automaten für reguläre Sprachen entwerfen. Es stellt sich heraus, dass die Klasse der regulären Automaten zu den größten Sprachenklassen gehört, für die optimale Programme effizient bestimmt werden können.

In der Praxis werden häufig nichtdeterministische Automaten benutzt. Wir zeigen, dass überraschenderweise jeder nichtdeterminische Automat einen äquivalenten deterministischen Automat besitzt. Im Abschnitt 1.2 haben wir bereits reguläre Ausdrücke kennengelernt und es stellt sich heraus, dass eine Sprache genau dann regulär ist, wenn die Sprache durch einen regulären Ausdruck dargestellt werden kann. Schliellich werden formale Sprachen häufig durch Grammatiken beschrieben: Deshalb werden wir reguläre Grammatiken einführen und zeigen, dass eine Sprache genau dann regulär ist, wenn sie eine reguläre Grammatik besitzt.

Reguläre Sprachen besitzen also eine Vielzahl äquivalenter Charakterisierungen. Neben diesen Eigenschaften des Maschinenmodells, bzw der Sprachenklasse, haben endliche Automaten trotz ihrer Einfachheit auch Anwendungen im Algorithmenentwurf. Hier besprechen wir zuerst kurz die lexikalische Analyse und untersuchen das Pattern Matching Problem in Beispiel 2.13.

Beispiel 2.3 Lexikalische Analyse.

Die lexikalische Analyse ist meistens die erste Phase bei der Compilierung eines vorgegebenen Programms. Das Programm wird dabei Anweisung nach Anweisung gelesen und in Tokenklassen aufgebrochen. (Beispiele für Tokenklassen sind Keywords, Konstanten, Variablen, arithmetische Operatoren, Vergleichsoperatoren, Klammern, Kommentare etc.) Zum Beispiel wird in der lexikalischen Analyse eine Anweisung der Form

```
if distance >= rate * (end - start) then distance = maxdistance;
```

aufgebrochen in die Tokenklassen

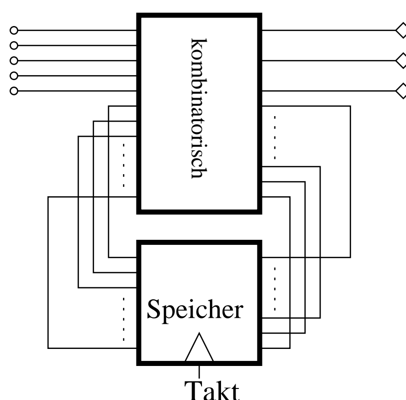
- Keywords (für `if` und `then`)
- Variablen (für `distance`, `rate`, `end`, `start`, `maxdistance`)
- Operatoren (für `*`, `-`, `=`)
- Vergleichsoperatoren (für `>=`)
- Klammern (für `(` und `)`)
- Semikolon (für `;`)

Weitere Phasen des Compilers bauen dann auf der Ausgabe der lexikalischen Analyse, also der Zuweisung von Tokenklassen, auf. Die lexikalische Analyse wird meist durch endliche Automaten implementiert, da die Erkennung von Tokenklassen durch „links-nach-rechts" Lesen des Programms möglich ist.

Neben den Eigenschaften des Maschinenmodells, bzw. der Sprachenklasse und den allerdings nur eingeschränkten algorithmischen Anwendungen sind endliche Automaten in der Modellierung wesentlich. Wir haben ein erstes Beispiel bereits in der Kindersicherung eines Fernsehers kennengelernt. Hier ist ein zweites Beispiel.

Beispiel 2.4 Modellierung sequentieller Schaltungen.

Bei einer sequentiellen Schaltung, im Gegensatz zu einer kombinatorischen Schaltung, hängt der Wert am Ausgang der Schaltung nicht nur von den aktuellen Eingangswerten ab, sondern auch von der Vorgeschichte, die sich im Inhalt der speichernden Bauteile niedergeschlagen hat. Der systematische Aufbau einer solchen Schaltung ist dabei wie folgt:



Wir modellieren die Schaltung durch einen Mealy Automaten. (Ein Mealy Automat ist ein endlicher Automat, der für jeden Zustandsübergang eine Ausgabe ausgibt.) Wir wählen als Eingabealphabet die Menge $\{0,1\}^n$, wenn n die Zahl der primären Eingänge ist. Wenn der Speicher s Bits speichert, dann benutzen wir die Zustandsmenge

$$Q = \{q_{b_1 \cdots b_s} \mid b_1, \ldots, b_s \in \{0,1\}\ \}.$$

Angenommen der Inhalt des Speichers ist b und eine Eingabe $x \in \{0,1\}^n$ liegt an. Wenn die Schaltung den neuen Speicher c sowie die Ausgabe y produziert, dann wählen wir den Zustandsübergang $q_b \stackrel{x}{\to} q_c$ und weisen diesem Zustandsübergang die Ausgabe y zu.

Eine Zustandsminimierung des Mealy Automaten führt zu einer Verkleinerung des Speichers, denn die Speichergröße ist logarithmisch in der Zustandszahl. Man beachte allerdings, dass diese Modellierung für grosses n nicht praktikabel ist und man wird diesen Ansatz, bzw. Erweiterungen wie OBDD's (ordered binary decision diagrams) nur für kleine Teilschaltungen anwenden können.

Wir werden das Minimierungsproblem im nächsten Kapitel angehen und lösen. Weitere Fragen, die wir in nachfolgenden Abschnitten besprechen, sind:

- Wann ist eine Sprache *nicht* regulär?
- Welche Sprachen werden von nichtdeterministischen endlichen Automaten akzeptiert?
- Welche — endliche Automaten betreffende — Fragen lassen sich effizient beantworten?

Letztendlich lernen wir alternative Charakterisierungen regulärer Sprachen mit regulären Grammatiken und regulären Ausdrücken kennen.

Aufgabe 7
Entwirf für folgende Sprachen deterministische endliche Automaten mit möglichst wenigen Zuständen:

(a) $L_n = \{w|\ w \in \{0,1\}^*$ und w hat mindestens n Einsen $\}$.
(b) $L_n = \{ww|\ w \in \{0,1\}^n\}$.
(c) $L = \{w|\ w \in \{0,1\}^*$ und die Anzahl der Einsen in w ist durch 7 teilbar$\}$.

Aufgabe 8
Entwirf einen deterministischen endlichen Automaten für folgende Sprachen:

$L_1 = \left\{w|w \in \{0,1,2,3,4\}^* \text{ und 3 teilt } \sum_{i=1}^{|w|} w_i 5^{i-1}.\right\}$

$L_2 = \left\{w|w \in \{0,1,2,3,4\}^* \text{ und 3 teilt } \sum_{i=1}^{|w|} w_i 5^{|w|-i}.\right\}$

Aufgabe 9
Die Sprache $(IF - STATEMENT)$ wird wie folgt rekursiv definiert:

$$IF - STATEMENT = \{if\} \circ EXPRESSION \circ \{then\} \circ STATEMENT$$

$EXPRESSION$ definieren wir als:

$$EXPRESSION = \{A, \ldots Z\}^+ \circ \{\geq 0\}$$

und $STATEMENT$ wird wie folgt definiert:

$$STATEMENT = IF - STATEMENT \cup \{A, \ldots, Z\}^+ \circ \{= 0\}$$

Zeige, dass die Sprache $IF - STATEMENT$ regulär ist.

Gibt es Schwierigkeiten im Nachweis der Regularität, wenn wir die rekursive Definition wie folgt verändern

$$IF-STATEMENT = \{if\} \circ EXPRESSION \circ \{then\} \circ STATEMENT \circ \{;\}$$

Aufgabe 10
Beweise oder **widerlege**:

(a) Zu jeder regulären Sprache L gibt es einen endlichen Automat A mit *genau einem* akzeptierendem Zustand und $L = L(A)$.

(b) Falls $L' \subseteq L$ und L regulär ist, dann ist auch L' regulär.

2.1 Nichtdeterministische endliche Automaten

Wir entwickeln jetzt nichtdeterministische endliche Automaten, um Sprachen *kurz beschreiben* zu können.

Definition 2.2 *Ein nichtdeterministischer endlicher Automat hat die folgenden Komponenten:*

- *Zustandsmenge* Q,
- *Eingabealphabet* Σ,
- *Anfangszustand* $q_0 \in Q$,
- $F \subseteq Q$ *als Menge der akzeptierenden Zustände*
- *sowie das Programm* δ*, das wir als Funktion*

$$\delta\colon Q \times \Sigma \to \mathcal{P}(Q)$$

auffassen, wobei $\mathcal{P}(Q)$ *die Potenzmenge von* Q *bezeichnet.*

Wenn ein deterministischer Automat im Zustand p den Buchstaben a liest, dann ist der Nachfolgezustand eindeutig festgelegt. Ein nichtdeterminischer Automat hingegen hat möglicherweise viele Optionen: Jeder Zustand in der *Zustandsmenge* $\delta(p, a)$ kommt als Nachfolgezustand in Frage! Ein nichtdeterministischer Automat hat also im Gegensatz zu einem deterministischen Automat viele verschiedene Berecnungen auf einer Eingabe w. Wann sollten wir sagen, dass w akzeptiert wird? Wir legen fest, dass w genau dann akzeptiert wird, wenn mindestens eine Berechnung erfolgreich ist, also auf einen akzeptierenden Zustand führt.

Definition 2.3 *Sei* $N = (Q, \Sigma, \delta, q_0, F)$ *ein nichtdeterministischer endlicher Automat.*

(a) Wir definieren die Fortsetzung

$$\delta\colon Q \times \Sigma^* \to \mathcal{P}(Q)$$

von Buchstaben auf Worte rekursiv wie folgt:

- $\delta(q, \varepsilon) = \{q\}$
- $\delta(q, wa) = \bigcup_{p \in \delta(q,w)} \delta(p, a)$

Hiermit drücken wir aus, dass ein Zustand $r \in Q$ durch Eingabe wa vom Zustand q aus „erreichbar“ ist, falls *ein Zustand p durch Eingabe w vom Zustand q erreichbar ist* und $r \in \delta(p, a)$ *gilt.*

(b) N akzeptiert w genau dann, wenn es einen akzeptierenden Zustand $p \in F$ *mit* $p \in \delta(q_0, w)$ *gibt.*

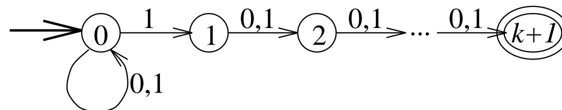
(c) $L(N) = \{w \in \Sigma^* \mid N \text{ akzeptiert } w\}$ *ist die von N akzeptierte (oder erkannte) Sprache.*

Beschreibungen regulärer Sprachen durch (deterministische) endliche Automaten erfordern manchmal „unangenehm“ viele Zustände, obwohl die Sprachen einfach sind.

Beispiel 2.5 Wie betrachten die Sprache

$L_k = \{0,1\}^* \cdot \{1\} \cdot \{0,1\}^k$.

L_k besteht also aus allen binären Worten, so dass der k+1.letzte Buchstabe eine Eins ist. Wir konstruieren einen sehr kleinen nichtdeterministischen Automaten, der die Position des $(k+1)$-letzten Bits *rät* und nachfolgend *verifiziert.* Dies führt auf das folgende Zustandsdiagramm:



Dem entspricht das Programm $\delta(⓪, 0) = \{⓪\}$, $\delta(⓪, 1) = \{⓪, ①\}$, $\ldots$, $\delta(ⓘ, 0) = \delta(ⓘ, 1) = \{(i+1)\}$ für $1 \leq i \leq k$ und $\delta((k+1), 0) = \delta((k+1), 1) = \emptyset$.

Man beachte, dass es durchaus erlaubt ist, den Automaten nur partiell zu definieren: Im obigen Beispiel haben wir keine Übergänge vom Zustand (k+1) aus definiert. Damit ist für alle $w \in \Sigma^*$

$$\delta((k+1), w) = \emptyset.$$

Wir zeigen andererseits, dass der Index von L_k mindestens 2^k beträgt. Damit muß also jeder endliche Automat für L_k mindestens 2^k Zustände und damit exponentiell mehr Zustände als unser nichtdeterministische Automat besitzen.

Es genügt zu zeigen, dass zwei beliebige, verschiedene Worte $u, v \in \{0,1\}^k$ nicht Nerode-äquivalent sind. Seien u, $v \in \{0,1\}^k$ also beliebige verschiedene Worte. Dann gibt es eine Position i mit $u_i \neq v_i$. Ohne Beschränkung der Allgemeinheit gelte

$$u_i = 0 \text{ und } v_i = 1.$$

Wir erhalten

$$v\,0^i = v_1 \cdots v_{i-1}\,1\,v_{i+1} \cdots v_k\,0^i \in L_k,$$

aber

$$u\,0^i = u_1 \cdots u_{i-1}\,0\,u_{i+1} \cdots u_k\,0^i \notin L_k.$$

Die beiden Worte u und v sind also nicht Nerode-äquivalent.

Sind denn nichtdeterministische Automaten überhaupt nützlich, d.h können wir das Wortproblem

> Überprüfe für einen gegebenen nichtdeterministischen Automaten N und eine Eingabe w, ob N die Eingabe w akzeptiert.

effizient lösen? Der Automat N rechnet nichtdeterministisch, aber wir müssen das Wortproblem mit einem deterministischen Algorithmus lösen!

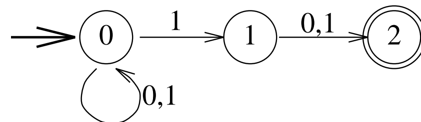
Aufgabe 11
Sei $N = (Q, \Sigma, \delta, q_0, F)$ ein nichtdeterministischer endlicher Automat und $w \in \Sigma^*$ eine Eingabe. Zeige, dass das Wortproblem für N und Eingabe w in Zeit $O(|w| \cdot |Q|^2)$ lösbar ist.

Das Wortproblem für nichtdeterministische Automaten ist also effizient lösbar und nichtdeterministische Automaten erlauben in einigen Fällen sehr viel kürzere Beschreibungen: Wo ist der Haken? Sind die von nichtdeterministischen Automaten erkannten Sprachen auch stets regulär? Da steckt der Haken nicht, wie wir im nächsten Abschnitt zeigen, der Haken steckt zum Beispiel in der Minimierung: Eine effiziente Minimierung nichtdeterministischer Automaten ist in aller Wahrschinlichkeit nicht mehr möglich.

2.1.1 Die Potenzmengenkonstruktion

Wir zeigen, wie äquivalente deterministische Automaten gebaut werden können und beginnen mit einem Beispiel.

Beispiel 2.6 Der nichtdeterministische Automat N mit Zustandsdiagramm



sei vorgegeben. N akzeptiert die Sprache $L_1 = \{0,1\}^* \cdot \{1\} \cdot \{0,1\}$. Unser Ziel ist die Konstruktion eines äquivalenten deterministischen Automaten A. Unsere Idee ist es, alle Berechnungen von N gleichzeitig zu simulieren. Demgemäß wählen wir Teilmengen der Zustandsmenge $Q = \{⓪, ①, ②\}$ als Zustände.

Ziel der Konstruktion: Wenn A die Eingabe w gelesen hat und sich im Zustand $s \subseteq Q$ befindet, dann fordern wir

$$s \overset{!}{=} \{p \in Q \mid \text{Es gibt eine Berechnung von } N \text{ für } w, \text{ die in } p \text{ endet}\}.$$

Die rechte Seite der Gleichung stimmt natürlich mit

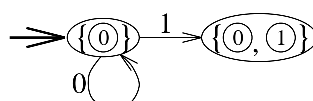
$$\delta_N(q_0, w)$$

überein. Wir geben jetzt die Konstruktion von A Schritt für Schritt an.

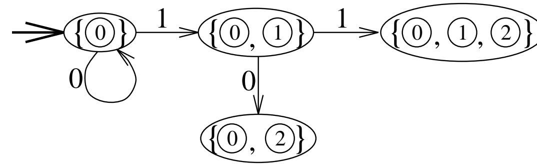
Schritt 1: Wir wählen $\{⓪\}$ als Anfangszustand von A. Mit Bit 0 kann N nur die Zustandsmenge $\{⓪\}$ erreichen, mit Bit 1 hingegen die Zustandsmenge $\{⓪, ①\}$. Wir setzen deshalb

$$\delta(\{⓪\}, 0) = \{⓪\} \text{ und } \delta(\{⓪\}, 1) = \{⓪, ①\}.$$

mit dem partiellen Zustandsdiagramm



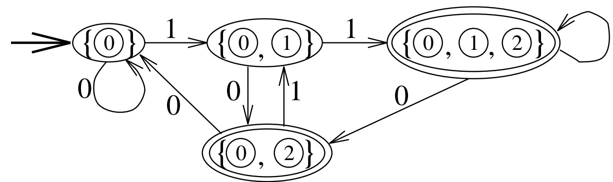
Schritt 2: Von {⓪ ,①} aus kann N mit Bit 0 (bzw. mit Bit 1) die Zustandsmenge {⓪,②} (bzw. {⓪,①,②}) erreichen. Deshalb setzen wir das Diagramm wie folgt fort:



Schritt 3: Wir setzen die Zustandsübergänge vom Zustand {⓪,②}. Mit Bit 0 wird nur der Zustand {⓪} erreicht, mit Bit 1 wird nur der Zustand {⓪,①} erreicht. Beide Zustände, {⓪} wie auch {⓪,①}, sind schon definiert.

Schritt 4: Wir setzen die Zustandsübergänge vom Zustand {⓪,①,②}. Mit Bit 0 (bzw. Bit 1) wird die Zustandsmenge {⓪,②} (bzw. {⓪,①,②} erreicht. Auch diese beiden Zustandsmengen haben wir bereits als Zustände definiert.

Damit erhalten wir insgesamt das Zustandsdiagramm



Tatsächlich wird die Sprache L_1 akzeptiert, wenn wir eine Zustandsmenge als akzeptierenden Zustand erklären, wann immer die Zustandsmenge einen akzeptierenden Zustand von N besitzt.

Die Potenzmengenkonstruktion, die wir in diesem Beispiel angewandt haben, kann für jeden nichtdeterministischen Automaten benutzt werden.

Satz 2.4 *Äquivalenz von deterministischen und nichtdeterministischen Automaten*

Sei N ein nichtdeterministischer Automat mit Komponenten Q_N, Σ, δ_N, q_0 und F_N. Der deterministische Automat A habe die Komponenten

- $Q_A = \{t \,|\, t \subseteq Q_N\}$,
- *Anfangszustand* $\{q_0\}$,
- $F_A = \{t \in Q_A \,|\, t$ *enthält einen Zustand aus* $F_N\}$
- δ_A *ist definiert durch*

$$\delta_A(t, a) = \Big\{p \in Q_N \,|\, \textit{Es gibt } q \in t \textit{ mit } p \in \delta_N(q, a)\Big\}.$$

Dann sind N und A äquivalent, das heißt es gilt $L(N) = L(A)$.

Beweis: Wir werden zeigen, dass

$$\delta_A(\{q_0\}, w) = \{p \in Q_N \,|\, \text{Es gibt eine Berechnung von } N \text{ für } w, \text{ die } p \text{ erreicht}\}. \tag{2.1}$$

Ist dies gezeigt, folgt

$$\begin{aligned} N \text{ akzeptiert } w &\Leftrightarrow \text{Es gibt } p \in F_N \text{ mit } p \in \delta_N(q_0, w) \\ &\Leftrightarrow \delta_A(\{q_0\}, w) \in F_A \\ &\Leftrightarrow A \text{ akzeptiert } w, \end{aligned}$$

und die Behauptung des Satzes ist gezeigt.

Wir beweisen (2.1) durch Induktion über die Länge von w.

Basis: $|w| = 0$.

Es ist $\delta_A(\{q_0\}, \varepsilon) = \{q_0\}$, aber auch alle Berechnungen von N für ε erreichen nur q_0.

Induktionsschritt: $|w| = n + 1$

Es gelte $w = w'a$ und $s' = \delta_A(\{q_0\}, w')$. Wir erhalten

$$\begin{aligned} \delta_A(\{q_0\}, w) &= \delta_A(\delta_A(\{q_0\}, w'), a) \\ &= \delta_A(s', a) \\ &= \{p \in Q \,|\, \text{Es gibt } q \in s' \text{ mit } p \in \delta_N(q, a)\} && (2.2) \\ &= \{p \in Q \,|\, \text{Es gibt eine Berechnung von } N \text{ für } w', \text{ die einen} \\ &\qquad \text{Zustand } q \in Q \text{ erreicht mit } p \in \delta_N(q, a)\} && (2.3) \\ &= \{p \in Q \,|\, \text{Es gibt eine Berechnung von } N \text{ für } w, \text{ die } p \\ &\qquad \text{erreicht}\}, && (2.4) \end{aligned}$$

und dies war zu zeigen. Gleichung (2.2) folgt gemäß Definition von δ_A. Gleichung (2.3) wendet die Induktionshypothese an, während Gleichung (2.4) die Definition von δ_N widerspiegelt. □

Bemerkung 2.1 A könnte überflüssige Zustände besitzen. Dies stellt kein Problem dar, wenn wir wie im Beispiel vorgehen, also nur Zustandsmengen betrachten, die nur aus erreichbaren Zuständen bestehen.

Aufgabe 12

Betrachte folgende Sprachen über dem Eingabealphabet $\Sigma = \{0, 1\}$:
$EQ_n = \{xy \mid x, y \in \{0,1\}^n,\ x = y\}$ und $NOT\text{-}EQ_n = \{xy \mid x, y \in \{0,1\}^n,\ x \neq y\}$.

Wieviele Zustände besitzt ein minimaler deterministischer endlicher Automat, der die Sprache $NOT\text{-}EQ_n$ erkennt? **Begründe** deine Antwort.

Aufgabe 13

Beschreibe einen nichtdeterministischen endlichen Automaten mit möglichst wenig Zuständen, der die Sprache $NOT\text{-}EQ_n$ erkennt.

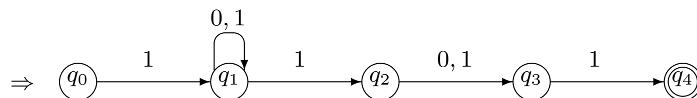
Zeige, dass ein nichtdeterministischer endlicher Automat, der die Sprache EQ_n erkennt *mindestens* 2^n Zustände benötigt.

Aufgabe 14

Wir betrachten nun Zwei-Wege-Automaten, also endliche Automaten, die den Kopf auch nach links bewegen können. Ein Zwei-Wege-Automat besteht aus den folgenden Komponenten: Dem Eingabealphabet Σ, der endlichen Menge Q der Zustände, dem Anfangszustand q_0, der Menge der akzeptierenden Zustände $F \subseteq Q$. Die Übergangsfunktion für Zwei-Wege-Automaten ist $\delta : Q \times \Sigma \to Q \times \{\mathsf{links}, \mathsf{rechts}\}$, wobei links und rechts die Richtung der Kopfbewegung beschreiben. Wenn ein Zwei-Wege-Automat in eine unendliche Schleife gerät oder die Eingabe nach links verläßt, wird die Eingabe nicht akzeptiert. Wenn der Zwei-Wege-Automat die Eingabe nach rechts verläßt, stoppt die Rechnung und die Eingabe wird genau dann akzeptiert, wenn sich der Automat in einem akzeptierenden Zustand befindet.

Beschreibe einen Zwei-Wege-Automaten mit möglichst wenigen Zuständen, der die Sprache EQ_n erkennt.

Aufgabe 15
Betrachte den folgenden nichtdeterministischen endlichen Automaten N über dem Eingabealphabet $\Sigma = \{0, 1\}$ mit der Zustandsmenge $Q = \{q_0, \ldots, q_4\}$, dem Anfangszustand q_0 und $F = \{q_4\}$:



Führe die Potenzmengenkonstruktion für N **aus** und **beschreibe** den resultierenden deterministischen Automaten (mit Ausnahme der *überflüssigen* Zustände!) durch sein Zustandsdiagramm.

Aufgabe 16
Für eine reguläre Sprache $L \subseteq \Sigma^*$ definieren wir

$$\text{Präfix}(L) = \{w \mid \text{ es gibt ein } v \in \Sigma^*, \text{ so dass } w \cdot v \in L \}.$$

Präfix(L) besteht also aus allen Präfixen von Wörtern aus L. Ist Präfix(L) regulär? Ist

$$\text{Suffix}(L) = \{w \mid \text{ es gibt ein } v \in \Sigma^*, \text{ so dass } v \cdot w \in L \}$$

regulär?

Aufgabe 17
Zeige oder **widerlege**: Wenn die unendlich vielen Sprachen $L_1, L_2, \ldots, L_n, L_{n+1}, \ldots$ alle regulär sind, dann ist auch die unendliche Vereinigung $\bigcup_{i=0}^{\infty} L_i$ regulär.

Aufgabe 18
Beweise oder **widerlege**: Sei L regulär. Dann stimmt die minimale Zustandszahl eines NFA A für L überein mit der kleinsten Zahl von Variablen einer regulären Grammatik G für L.

Aufgabe 19
Das Minimierungsproblem für NFA's ist mit aller Wahrscheinlichkeit nicht effizient lösbar: alle bekannten Minimierungsalgorithmen benötigen eine Laufzeit, die exponentiell in der Anzahl der Zustände des zu minimierenden NFA ist.

Auch die Bestimmung der minimalen Zustandszahl ist ein komplexes Problem, das sich aber in einigen Fällen lösen lässt.

(a) **Zeige**, dass jeder NFA für

$$L_m = \{a^n : n \in N \text{ und } n \text{ ist durch } m \text{ teilbar}\}$$

mindestens m Zustände besitzt.

(b) **Beweise** oder **widerlege**: Wenn es NFA's für L_1 bzw. L_2 mit höchstens n_1 bzw. n_2 Zuständen gibt, dann gibt es einen NFA für $L_1 \cap L_2$ mit höchstens $n_1 + n_2$ Zuständen.

2.2 Das Pumping-Lemma

Wir möchten Techniken entwickeln, um die Grenzen der Berechnungskraft endlicher Automaten festzustellen; d.h. um feststellen zu können, wann eine Sprache nicht regulär ist.

Satz 2.5 (Pumping-Lemma) *Sei L regulär. Dann gibt es eine Pumpingkonstante N, so dass jedes Wort $z \in L$ mit $|z| \geq N$ eine Zerlegung mit den Eigenschaften*

(a) $z = uvw$, $|uv| \leq N$ und $|v| \geq 1$,

(b) $uv^iw \in L$ für jedes $i \geq 0$

besitzt. (Wenn Worte der Sprache lang genug sind, dann gibt es ein nicht-leeres Teilwort v, das „aufgepumpt" ($i \geq 1$) oder „abgepumpt" ($i = 0$) werden kann.)

Beweis: Da L regulär ist, gibt es einen endlichen Automaten A, der L akzeptiert. Sei q_0 Anfangszustand und Q Zustandsmenge von A. Wir wählen

$$N = |Q|$$

als Pumping-Konstante. Sei $z \in L$ mit $z = z_1 z_2 \cdots z_s$ und $|z| = s \geq N$ beliebig gewählt. Der Automat A durchläuft die Zustandsfolge

$$q_0 \overset{z_1}{\Rightarrow} q_1 \overset{z_2}{\Rightarrow} q_2 \overset{z_3}{\Rightarrow} \cdots \overset{z_N}{\Rightarrow} q_N \overset{z_{N+1}}{\Rightarrow} \cdots \overset{z_s}{\Rightarrow} q_s,$$

wobei q_s ein akzeptierender Zustand ist. Nachdem A den Präfix $z_1 z_2 \cdots z_N$ von z gelesen hat, hat A genau $N + 1$ Zustände durchlaufen. Ein Zustand q_i muß also zweimal aufgetreten sein und es gibt i, j, $j \neq i$, mit $q_i = q_j$. Demgemäß wählen wir die Zerlegung (falls $i < j$)

$$u = z_1 \cdots z_i,\ v = z_{i+1} \cdots z_j,\ w = z_{j+1} \cdots z_s$$

Dann ist $|uv| \leq N$ und $|v| \geq 1$. Weiterhin gilt

$$\delta(q_0, u) = q_i = q_j = \delta(q_0, uv)$$

und deshalb akzeptiert A ebenfalls die Worte

$$uw, uv^2w, uv^3w, \ldots, uv^iw, \ldots$$

□

Wie zeigt man, dass eine Sprache L nicht regulär ist? Wir müssen das Pumping Lemma falsifizieren. Wenn L regullär ist, dann

- gibt es eine (uns unbekannte) Pumpingkonstante N,
- so dass für jedes Wort $z \in L$ mit $|z| \geq N$
- eine Zerlegung $z = uvw$ mit $|uv| \leq N$ und $|v| \geq 1$ gibt, so dass

  $$uv^iw \in L$$

  für alle $i \geq 0$ gilt.

Wie zeigt man also, dass die Sprache L nicht regulär ist?

- Für jedes mögliche N
- müssen wir ein Wort $z \in L$ mit $|z| \geq N$ konstruieren,
- so dass für jede mögliche Zerlegung $z = uvw$ mit $|uv| \leq N$ und $|v| \geq 1$

  $$uv^iw \notin L$$

  für mindestens ein $i \geq 0$ gilt.

Wir müssen also für jede mögliche Zerlegung $z = uvw$ ein i finden, so dass $uv^iw \notin L$. Ist dies gelungen, haben wir einen Widerspruch zur Regularität von L erhalten.

Beispiel 2.7 Wir zeigen, dass die Sprache aller Palindrome über $\{0,1\}$, also

$$L = \left\{ w \in \{0,1\}^* \mid w = w^{\text{reverse}} \right\}$$

nicht regulär ist. Wir folgen dem obigen Rezept.

- N sei die unbekannte Pumping-Konstante.
- Wir wählen $z = 0^N \, 1 \, 0^N$ (und sichern damit $|z| \geq N$).
- Für **jede** Zerlegung $z = uvw$ mit $|uv| \leq N$ und $|v| \geq 1$ müssen wir $uv^iw \notin L$ für mindestens ein i nachweisen.

Offensichtlich ist uv ein Präfix von 0^N, denn wir wissen, dass $|uv| \leq N$ gilt. Dann gibt es aber ein k mit $v = 0^k$ und

$$uv^2w = 0^{N+k} \, 1 \, 0^N \notin L.$$

Die Pumping-Eigenschaft zeigt, dass die Klasse der regulären Sprachen sehr eingeschränkt ist. **Aber**:

Aufgabe 20
Zeige, dass **jede** endliche Menge regulär ist.

Aufgabe 21
Wir beweisen mit Hilfe des Pumping Lemmas, dass die endliche Sprache $L = \{a,b\}^{100}$ nicht regulär ist.

> Nehmen wir an L sei regulär. Aus dem Pumping Lemma folgt, dass es für das Wort a^{100} eine Zerlegung $a^{100} = xyz$ mit $|xy| \leq$Pumpingkonstante und $|y| \geq 1$ gibt, so dass jedes Wort xy^kz (mit $k \geq 0$) auch zu L gehört. Wir erhalten einen Widerspruch zur Annahme, dass die Sprache L endlich ist, denn es gibt unendlich viele Strings der Form xy^kz.

Finde den Fehler im Beweis.

Aufgabe 22
Zeige mit Hilfe des Pumping Lemmas, dass die folgenden Sprachen nicht regulär sind:

(a) $L = \{w \in \{0,1\}^* \mid w$ hat genau so viele Nullen wie Einsen $\}$.

(b) $L = \{ww^{reverse} \mid w \in \{0,1\}^*\}$.

Aufgabe 23
Beweise das verallgemeinerte Pumping Lemma. Die zu beweisende Aussage ist wie folgt:

Sei L eine reguläre Sprache. Dann **gibt** es eine Konstante n, so dass für **jedes** $z \in L$ mit **vorgegebener** Partition $z = tyx$ mit $|y| = n$ eine Zerlegung $y = uvw$ mit $|v| \geq 1$ gefunden werden kann, so dass $tuv^iwx \in L$ für alle $i \geq 0$.

(Mit anderen Worten, das verallgemeinerte Pumping Lemma erlaubt ein „Fenster“ der Länge n über die Eingabe zu schieben, so dass innerhalb dieses Fensters „gepumpt“ werden kann.)

Aufgabe 24
Wende dieses verallgemeinerte Pumping Lemma an, um zu zeigen, dass die Sprache $L = \{1^k \mid k \geq 0\} \cup \{0^j 1^{(k^2)} \mid j \geq 1,\, k \geq 0\}$ nicht regulär ist. (L enthält alle Wörter, die mit mindestens einer Null beginnen, gefolgt von einer quadratischen Anzahl von Einsen.)

Warum kann mit Hilfe des Pumping Lemmas der Vorlesung **nicht** gezeigt werden, dass L von keinem endlichen Automaten akzeptiert wird?

(Das Pumping Lemma liefert somit keine hinreichende Bedingung für die Regularität von Sprachen.)

Aufgabe 25
Zeige, dass die wie folgt definierte Sprache über dem Eingabealphabet $\Sigma = \{0,1\}$ nicht regulär ist:

$$L = \{w \in \{0,1\}^* \mid w \text{ hat eine Zerlegung der Form } w = x_1 \dots x_n\, y_1 \dots y_n\, z_0, z_1 \dots z_n \text{ für ein } n \in \mathbb{N} \text{ und es gilt } \sum_{i=0}^{n} z_i 2^{n-i} = \sum_{i=1}^{n} x_i 2^{n-i} + \sum_{i=1}^{n} y_i 2^{n-i}\}$$

(Dies soll salopp formuliert heißen: Ein Wort $w \in \{0,1\}^*$ liegt in L, wenn es eine "korrekte Binäraddition" beschreibt.)

2.3 Minimierung endlicher Automaten

Das Ziel dieses Kapitels ist die Konstruktion eines effizienten Algorithmus zur Minimierung endlicher Automaten.

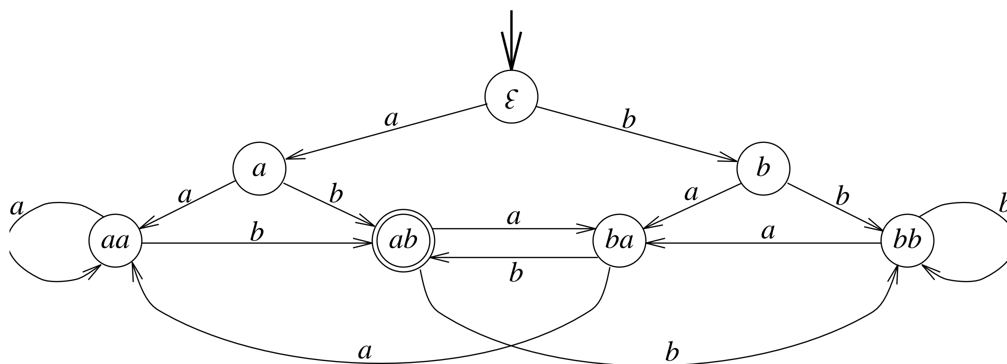
Beispiel 2.8 Wir behaupten, dass die Sprache

$$L = \{a,b\}^*\, ab$$

regulär ist. Wir entwerfen dazu einen endlichen Automaten, der sich die beiden letzten gelesenen Buchstaben in seinen Zuständen merkt. Unser Automat hat die Komponenten

- $\Sigma = \{a,b\}$
- $Q = \{\varepsilon, a, b, aa, ab, ba, bb\}$
- $q_0 = \varepsilon$ und $F = \{ab\}$.

Sein Programm beschreiben wir durch das Zustandsdiagramm

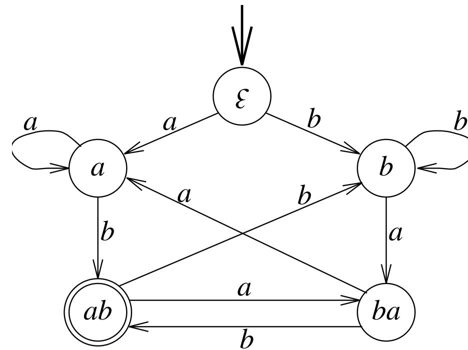


Dieser Automat merkt sich tatsächlich die beiden letzten Buchstaben, denn es gilt

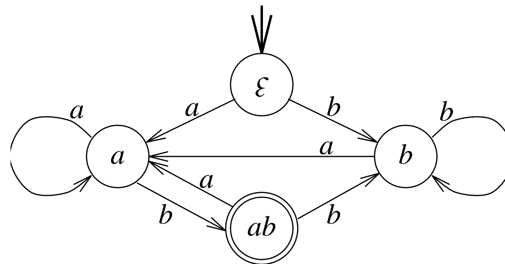
$$\delta(\varepsilon, w) = z \quad \Leftrightarrow \quad (|w| \leq 1 \text{ und } w = z) \text{ oder } (|z| = 2 \text{ und } w = w'z)$$

Diese Aussage kann man zum Beispiel durch Induktion über die Länge der Eingabe w verifizieren. Ist dieser Automat auch minimal, d.h. hat der Automat die kleinste Anzahl von Zuständen unter allen Automaten, die $\Sigma^* ab$ akzeptieren?

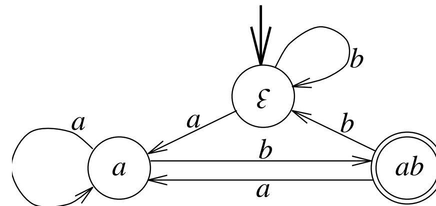
Die Antwort ist nein! Ist zum Beispiel a der zuletzt gelesene Buchstabe, dann ist es uninteressant, was der vorletzte Buchstabe war. Dies äußert sich im Zustandsdiagramm daran, dass die Zustände a und aa unter dem Buchstaben a wie auch unter dem Buchstaben b dieselben Nachfolgerzustände (nämlich aa und ab) erreichen. Die Zustände a und aa können also verschmolzen werden. Das Gleiche trifft auch für die Zustände b und bb zu: Auch sie können verschmolzen werden. Wir erreichen also das neue Zustandsdiagramm



Jetzt stellen sich die Zustände a und ba als äquivalent heraus, da beide Zustände nicht akzeptierend und die Nachfolgezustände identisch sind. Nach Verschmelzung ergibt sich das Zustandsdiagramm



und plötzlich stellen sich ε und b als äquivalent heraus. Nach ihrer Verschmelzung haben wir somit den Automaten



Ist dieser Automat minimal?

Gegeben sei ein endlicher Automat $A = (Q, \Sigma, \delta, q_0, F)$. Alle Zustände, die vom Anfangszustand q_0 aus nicht erreichbar sind, heißen überflüssig.

Als ersten Schritt des Minimierungsalgorithmus entfernen wir alle überflüssigen Zustände. Dies können wir zum Beispiel durch eine Tiefensuche erreichen, die im Startzustand beginnt. Alle nicht besuchten Zustände sind überflüssig. (Da der Automat $|Q|$ Knoten mit jeweils $|\Sigma|$ ausgehenden Kanten besitzt, findet eine Tiefensuche alle nicht-überflüssigen Zustände in Zeit $O(|Q| \cdot |\Sigma|)$.)

Wir nehmen nun an, dass der zu minimierende Automat keine überflüssigen Zustände besitzt. Wir haben bereits im letzten Beispiel gesehen, dass ein Automat ohne überflüssige Zustände nicht minimal sein muß.

Im letzten Beispiel haben wir auch einen ersten Ansatz zum Zusammenfassen von Zuständen kennengelernt: Wir haben zwei Zustände p und q äquivalent genannt, wenn p und q beide zu F

(oder beide zum Komplement von F) gehören und wenn $\delta(p,a) = \delta(q,a)$ für alle Buchstaben $a \in \Sigma$ gilt. Äquivalente Zustände können wir offensichtlich zu einem einzigen Zustand verschmelzen.

Wird dieses Verfahren stets zu einem minimalen Automaten führen?

Die Antwort ist leider *nein*.

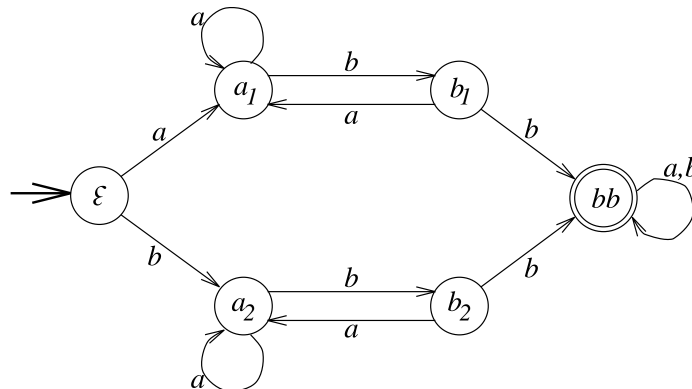
Beispiel 2.9 Wir betrachten die Sprache

$$L = \{a,b\}^+ \, bb \, \{a,b\}^*.$$

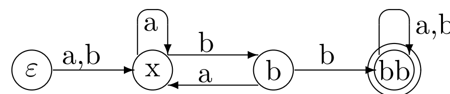
L kann von dem Automaten mit

- Zustandsmenge $Q = \{\varepsilon, a_1, a_2, b_1, b_2, bb\}$,
- Anfangszustand ε,
- $F = \{bb\}$

und Zustandsdiagramm



akzeptiert werden. Beachte, dass keine zwei Zustände (gemäß unserer Definition) äquivalent sind. Aber dieser Automat ist nicht minimal: Die Zustände a_1, a_2 und b_1, b_2 können verschmolzen werden, und der folgende Automat ist äquivalent zu



Unsere Definition äquivalenter Zustände ist also zu restriktiv: Die Zustände a_1 und a_2 hätten als äquivalent bezeichnet werden sollen, denn von a_1 *wie auch* von a_2 erreichen wir genau dann einen akzeptierenden Zustand, wenn aufeinanderfolgende b's erhalten werden. Von diesem Standpunkt aus gesehen unterscheiden sich a_1 und a_2 also nicht.

Definition 2.6 *Die Verschmelzungsrelation.*

$A = (Q, \Sigma, \delta, q_0, F)$ sei ein endlicher Automat. Wir sagen, dass die Zustände $p, q \in Q$ äquivalent sind (geschrieben $p \equiv_A q$) genau dann, wenn für alle $w \in \Sigma^*$

$$\delta(p,w) \in F \quad \Leftrightarrow \quad \delta(q,w) \in F.$$

gilt. Mit anderen Worten, wir nennen Zustände $p, q \in Q$ äquivalent, wenn p und q sich nicht unterscheiden, wenn es „ums Akzeptieren geht“.

Kehren wir zurück zum letzten Beispiel. Wir erhalten $a_1 \equiv_A a_2$ und $b_1 \equiv_A b_2$. Können wir weitere Paare äquivalenter Zustände entdecken? Diesmal ist die Antwort nein; denn

Behauptung: $\varepsilon \not\equiv_A x$

Dies folgt, da $\delta(x, bb) \in F$, aber $\delta(\varepsilon, bb) \notin F$.

Behauptung: $\varepsilon \not\equiv_A b$

denn $\delta(\varepsilon, b) \notin F$, aber $\delta(b, b) \in F$.

Behauptung: $x \not\equiv_A b$

denn $\delta(x, b) \notin F$, aber $\delta(b, b) \in F$.

Der Zustand *bb* ist trivialerweise nicht äquivalent zu einem der anderen Zustände, da er der einzige akzeptierende ist. (Somit gilt z.B. $x \not\equiv_A bb$, denn $\delta(x, \varepsilon) = x \notin F$ aber $\delta(bb, \varepsilon) = bb \in F$.)

Lemma 2.7 *Für jeden endlichen Automaten A ist die Relation* $\equiv_A$

- *reflexiv, d.h. es ist* $q \equiv_A q$ *für alle Zustände q von A,*
- *symmetrisch, d.h. für alle Zustände p und q von A gilt* $p \equiv_A q$ *genau dann, wenn* $q \equiv_A p$ *gilt und*
- *transitiv, d.h. für alle Zustände p, q und r folgt aus* $p \equiv_A q$ *und* $q \equiv_A r$ *stets* $p \equiv_A r$.

Also ist $\equiv_A$ *eine Äquivalenzrelation.*

Beweis: Übungsaufgabe. □

2.3.1 Der Äquivalenzklassenautomat

Können wir äquivalente Zustände verschmelzen, und wenn ja, wie? Die folgende Definition gibt das Rezept:

Definition 2.8 $A = (Q, \Sigma, \delta, q_0, F)$ *sei ein endlicher Automat.*

(a) Für Zustand $p \in Q$ *bezeichnet*

$$[p] = \left\{ q \in Q \,|\, p \equiv_A q \right\}$$

die Äquivalenzklasse von p.

(b) Wir definieren den Äquivalenzklassenautomaten A′ für A. A′ besitzt

- *die Zustandsmenge* $Q' = \left\{ [p] \,|\, p \in Q \right\}$,
- *den Anfangszustand* $[q_0]$,
- *und* $F' = \left\{ [p] \,|\, p \in F \right\}$ *als Menge der akzeptierenden Zustände.*

Das Programm δ' *definieren wir durch*

$$\delta'([p], a) = [\delta(p, a)].$$

Funktioniert diese Definition stets? Das heißt, ist der Automat A' äquivalent zu A? Ist der Automat überhaupt wohldefiniert?

Satz 2.9 *Der Äquivalenzklassenautomat A' ist wohldefiniert und äquivalent zu A, d.h. es gilt $L(A) = L(A')$).*

Beweis: Warum ist die Wohldefiniertheit von A' nicht sofort klar? Wir haben

$$\delta'([p], a) = [\delta(p, a)]$$

gesetzt. Angenommen, p und q sind äquivalent. Dann haben wir ein ganz großes Problem, falls

$$[\delta(p, a)] \neq [\delta(q, a)]$$

gilt. In diesem Fall hängt unsere Definition nämlich von der Wahl des Repräsentanten der Äquivalenzklasse $[p]$ ab! Aber wir brauchen uns keine Sorgen zu machen: Wenn p und q äquivalent sind, dann gilt für alle $w \in \Sigma^*$,

$$\delta(p, w) \in F \quad \Leftrightarrow \quad \delta(q, w) \in F.$$

Insbesondere folgt für alle $w \in \Sigma^*$,

$$\delta(p, aw) \in F \quad \Leftrightarrow \quad \delta(q, aw) \in F.$$

Es ist

$$\begin{aligned} \delta(p, aw) &= \delta(\delta(p, a), w) \quad \text{und} \\ \delta(q, aw) &= \delta(\delta(q, a), w). \end{aligned}$$

Damit folgt für alle $w \in \Sigma^*$

$$\delta(\delta(p, a), w) \in F \quad \Leftrightarrow \quad \delta(\delta(q, a), w) \in F$$

und wir erhalten

$$\delta(p, a) \equiv_A \delta(q, a),$$

als Konsequenz.

Wir haben Wohldefiniertheit gezeigt, sind A und A' aber auch äquivalent? Wir vergleichen Berechnungen von A und A'. Sei $w \in \Sigma^*$ eine Eingabe.

Behauptung:

$$\delta'([q_0], w) = [\delta(q_0, w)].$$

Wir führen einen induktiven Beweis über die Länge von w. Aus der Definition folgt $\delta'([q_0], a) = [\delta(q_0, a)]$ für jeden Buchstaben a: Wir haben die Induktionsbasis nachgewiesen. Für den Induktionsschritt beachten wir zuerst, dass

$$\delta'([q_0], w_1 \cdots w_n) = \delta'\Big(\delta'([q_0], w_1 \cdots w_{n-1}), w_n\Big)$$

aus der Arbeitsweise des endlichen Automaten A folgt. Wir können induktiv annehmen, dass $\delta'([q_0], w_1 \cdots w_{n-1}) = [\delta(q_0, w_1 \cdots w_{n-1})]$ gilt. Wenn also $p = \delta(q_0, w_1 \cdots w_{n-1})$ ist, dann folgt

$$\begin{aligned} \delta'([q_0], w_1 \cdots w_n) &= \delta'\Big(\delta'([q_0], w_1 \cdots w_{n-1}), w_n\Big) \\ &= \delta'([p], w_n) \\ &= [\delta(p, w_n)], \end{aligned}$$

wobei wir im letzten Schritt wieder die Wohldefiniertheit angewandt haben. Und das war zu zeigen. □

Wir zeigen zuerst die Inklusion $L(A) \subseteq L(A')$. Sei $w \in L(A)$. Dann ist $\delta(q_0, w) \in F$ und als Konsequenz folgt $\delta'(q_0, w) = [\delta(q_0, w)] \in F'$.

Nun zum Beweis der umgekehrten Inklusion $L(A') \subseteq L(A)$. Sei $w \in L(A')$. Dann ist $\delta'([q_0], w) = [\delta(q_0, w)]$ und $[\delta(q_0, w)] \in F'$. Wir müssen also zeigen, dass $\delta(q_0, w) \in F$ gilt.

Nach Definition von F' gilt $\delta(q_0, w) \equiv_A p$ für einen Zustand $p \in F$. Gemäß der Definition der Äquivalenzrelation $\equiv_A$ folgt aber insbesondere

$$\delta(\delta(q_0, w), \varepsilon) \in F \quad \Leftrightarrow \quad \delta(p, \varepsilon) \in F.$$

Das letztere ist der Fall, da $p \in F$. Und somit ist $\delta(\delta(q_0, w), \varepsilon) \in F$, und $\delta(q_0, w)$ muß ein akzeptierender Zustand sein. Also ist $w \in L(A)$. □

Wir haben das Minimierungsproblem gelöst, wenn

- wir den Äquivalenzklassenautomaten effizient berechnen können und
- zeigen können, dass der Äquivalenzautomat minimal ist.

Wir beginnen mit dem Problem der effizienten Berechnung des Äquivalenzklassenautomats.

Anstatt Äquivalenz von Zuständen zu zeigen, versuchen wir, Nicht-Äquivalenz nachzuweisen. Insbesondere versuchen wir für jedes Paar $\{p, q\}$ von nicht-äquivalenten Zuständen einen *Zeugen* $w \in \Sigma^*$ zu finden. Dieser Zeuge muß die Eigenschaft

(1) $\delta(p, w) \in F$ und $\delta(q, w) \notin F$ oder

(2) $\delta(p, w) \notin F$ und $\delta(q, w) \in F$

besitzen. Die folgende Beobachtung erleichtert das Auffinden von Zeugen: Wenn die Zustände $\delta(p, a)$ und $\delta(q, a)$ nicht äquivalent sind (mit dem Zeugen w), dann sind auch p und q nicht äquivalent (mit dem Zeugen aw). (Denn wenn zum Beispiel $\delta(\delta(p, a), w) \in F$ und $\delta(\delta(q, a), w) \notin F$, dann ist $\delta(p, aw) = \delta(\delta(p, a), w) \in F$ und $\delta(q, aw) = \delta(\delta(q, a), w) \notin F$.)

Wir können jetzt einen Algorithmus zur Bestimmung äquivalenter Zustände bereits skizzieren. Der Algorithmus beginnt mit der Markierung aller Paare

$$\{p, q\}$$

mit $p \in F$ und $q \notin F$ (bzw. $p \notin F$ und $q \in F$) als nicht-äquivalent. Alle solchen Paare bestehen offensichtlich aus nicht-äquivalenten Zuständen, da

$$\delta(p, \varepsilon) \in F \text{ und } \delta(q, \varepsilon) \notin F \text{ (bzw. } \delta(p, \varepsilon) \notin F \text{ und } \delta(q, \varepsilon) \in F\text{) ist.}$$

Jetzt wird versucht, alle Konsequenzen markierter Paare zu bestimmen. Das heißt,

> wenn $p = \delta(p', a)$ und $q = \delta(q', a)$ markiert sind, dann dürfen wir auch p' und q' markieren.

Wie können wir einen solchen Markierungsprozeß leicht durchführen? Wir bauen einen Graphen G. Die Knoten von G entsprechen Paaren verschiedener Zustände. Wir fügen eine Kante

$$\{p, q\} \rightarrow \{p', q'\}$$

genau dann ein, wenn es einen Buchstaben $a \in \Sigma$ gibt mit

$$p = \delta(p', a) \quad \text{und} \quad q = \delta(q', a).$$

Jetzt läßt sich der Markierungsprozeß durchführen, indem wir eine Tiefensuche auf den anfänglich markierten Paaren ausführen! Die formale Beschreibung folgt:

Initialisierung: Bestimme die Adjazenzlisten-Darstellung von G. Markiere alle Paare $\{p, q\}$ für die entweder ($p \in F$ und $q \notin F$) oder ($p \notin F$ und $q \in F$) gilt.

Berechnung: Führe eine Tiefensuche durch, die

> von markierten (und bisher nicht besuchten) Knoten aus aufgerufen wird und
>
> jeden besuchten Knoten markiert.

Ausgabe: Die Äquivalenzklasse von p wird als die Menge

$$\Big\{ q \in Q \mid \{p, q\} \text{ wurde nicht markiert} \Big\}$$

ausgegeben.

Satz 2.10 *Der obige Algorithmus bestimmt alle Äquivalenzklassen in Zeit*

$$O\left(|Q|^2 \cdot |\Sigma|\right)$$

Beweis: Wir beginnen mit einem Korrektheitsbeweis. Zuerst beachten wir, dass für äquivalente Zustände p und q das Paar $\{p, q\}$ nicht markiert wird, denn unser Markierungsprozeß führt von nicht-äquivalenten Zuständen auf nicht-äquivalente Zustände. Es ist also nur zu klären, dass jedes Paar nicht-äquivalenter Zustände auch tatsächlich markiert wird.

Angenommen, es gibt nicht-äquivalente Zustände p und q, die nicht markiert werden. Wir wählen unter allen nicht-äquivalenten, nicht markierten Zuständen ein Paar $\{p', q'\}$ mit kürzestem Zeugen w. Es gelte zum Beispiel $\delta(p', w) \in F$ und $\delta(q', w) \notin F$. Offensichtlich ist $w \neq \varepsilon$, denn Paare mit Zeugen ε werden anfangs markiert.

Für $w = aw'$ sind auch die Zustände $\delta(p', a)$ und $\delta(q', a)$ nicht-äquivalent, denn $\delta(\delta(p', a), w') \in F$ und $\delta(\delta(q', a), w') \notin F$. Diese beiden Zustände besitzen aber einen kürzeren Zeugen w' und sind deshalb markiert. Aber dann markiert unsere Markierungsstrategie auch p' und q', ein Widerspruch zur Annahme.

Wir kommen zur Laufzeitbestimmung. Die *Initialisierung* gelingt in Zeit $O\left(|Q|^2 \cdot |\Sigma|\right)$, denn der Graph G besitzt $\binom{|Q|}{2} = \Theta\left(|Q|^2\right)$ Knoten und höchstens

$$\binom{|Q|}{2} \cdot |\Sigma|$$

Kanten. Die *Berechnung* erfolgt durch Tiefensuche. Da Tiefensuche in Zeit linear in der Summe von Knotenanzahl und Kantenanzahl verläuft, benötigt die Berechnung also höchstens

$$O\left(|Q|^2 + |Q|^2 \cdot |\Sigma|\right) = O\left(|Q|^2 \cdot |\Sigma|\right)$$

viele Schritte. Die *Ausgabe* kann schließlich in Zeit $O\left(|Q|^2\right)$ erbracht werden. $\square$

Wir sind der Lösung des Minimierungsproblems also einen wesentlichen Schritt näher gekommen. Wir kommen jetzt zum letzten, aber wichtigsten Problem: Ist der Äquivalenzklassenautomat minimal?

2.3.2 Die Nerode-Relation

Wie könnte ein minimaler Automat A für eine Sprache L aussehen? Wir repräsentieren einen Zustand p von A mit einem beliebigen Wort $x_p \in \Sigma^*$, das p vom Anfangszustand aus erreicht, d.h. für das $\delta(q_0, x_p) = p$ gilt. Aus der Diskussion des Äquivalenzklassenautomaten A' von A haben wir gelernt, dass A nicht minimal ist, wenn A zwei oder mehr äquivalente Zustände besitzt. Mit anderen Worten, für je zwei verschiedene Zustände p und q darf

$$x_p w \in L \quad \Leftrightarrow \quad x_q w \in L$$

nicht für alle Worte $w \in \Sigma^*$ gelten.

Definition 2.11 *Sei L eine Sprache über Σ. Die Nerode-Relation $\equiv_L$ für Worte in Σ^* wird wie folgt definiert: $x \equiv_L y$ gilt genau dann, wenn für alle Worte $w \in \Sigma^*$ gilt*

$$xw \in L \quad \Leftrightarrow \quad yw \in L.$$

Den Index von L definieren wir als die Anzahl der Äquivalenzklassen von $\equiv_L$.

Hier ist der Clou: Wir bauen aus der Nerode-Relation einen minimalen endlichen Automaten, den Nerode-Automaten, indem wir die Äquivalenzklassen von $\equiv_L$ als Zustandsmenge benutzen.Damit ist der Index von L nichts anderes als die Anzahl der Zustände eines minimalen endlichen Automaten. Wir werden dann den Äquivalenzklassenautomaten mit dem Nerode-Automaten vergleichen und feststellen, dass beide identisch sind. Und damit ist auch das letzte Problem aus dem Wege geräumt.

Zuerst zeigen wir, dass der Index von L eine untere Schranke für die Zustandsanzahl eines Automaten A mit $L = L(A)$ ist.

Satz 2.12 *Es gelte $L = L(A)$ für einen endlichen Automaten A.*

(a) Wenn Worte x und y auf denselben Zustand von A führen, d.h. wenn $\delta(q_0, x) = \delta(q_0, y)$ gilt, dann ist $x \equiv_L y$.

(b) Sei Q die Zustandsmenge von A. Dann gilt $|Q| \geq$ Index von L.

Beweis (a): Wir müssen zeigen, dass

$$xw \in L \quad \Leftrightarrow \quad yw \in L$$

für alle $w \in \Sigma^*$ gilt. Da $\delta(q_0, x) = \delta(q_0, y)$, folgt $\delta(q_0, xw) = \delta(q_0, yw)$ für jedes Wort w. Somit gilt

$$\begin{aligned} xw \in L &\Leftrightarrow \delta(q_0, xw) \in F \\ &\Leftrightarrow \delta(q_0, yw) \in F \\ &\Leftrightarrow yw \in L. \end{aligned}$$

(b) Wir erhalten zwei Äquivalenzrelationen auf allen Worten über Σ, nämlich die Nerode-Relation $\equiv_L$ und eine zweite Äquivalenzrelation R_A, die zwei Worte x und y äquivalent nennt, wenn $\delta(q_0, x) = \delta(q_0, y)$ gilt. Beachte, dass die Anzahl der Äquivalenzklassen von R_A mit der Anzahl der Zustände von A übereinstimmt.

Wenn x und y bezüglich R_A äquivalent sind, wenn also x und y denselben Endzustand in A erreichen, dann wissen wir nach Teil (a), dass x und y Nerode-äquivalent sind. Also sind alle Nerode-Äquivalenzklassen Vereinigungen von Äquivalenzklassen von R_A. Dann besitzt R_A aber mindestens so viele Äquivalenzklassen wie die Nerode-Relation und $|Q|$ ist mindestens so groß wie der Index von L. □

Beispiel 2.10 Wir betrachten die Sprache

$$L_m = \{w \in \{0,1\}^* \mid \sum_i w_i \text{ ist durch } m \text{ teilbar}\}.$$

Wie sehen die Äquivalenzklassen der Nerode-Relation von L_m aus? Es ist

$$\begin{aligned} x \equiv_{L_m} y &\Leftrightarrow \text{ fuer alle } w \in \{0,1\}^* : (xw \in L_m \Leftrightarrow yw \in L_m) \\ &\Leftrightarrow \sum_i x_i \equiv \sum_i y_i \mod m. \end{aligned}$$

Also hat $\equiv_{L_m}$ genau m Äquivalenzklassen, die den m Restklassen modulo m entsprechen: Die Repräsentanten dieser Restklassen sind $\varepsilon, 1, 1^2, \ldots, 1^{m-1}$. Wir bauen jetzt einen minimalen endlichen Automaten mit Zustandsmenge $[\varepsilon], [1], [1^2], \ldots, [1^{m-1}]$, wobei $[x]$ die Nerode-Äquivalenzklasse des Worts x bezeichnet.

- $[\varepsilon]$ ist der Startzustand: Bisher haben wir noch keine Eins gelesen.
- $[\varepsilon]$ ist auch unser einziger akzeptierender Zustand, denn für jeden anderen Zustand $[1^i]$ ist die Anzahl i der gelesenen Einsen nicht durch m teilbar.
- Mit Hilfe der Überführungsfunktion δ zählen wir die Anzahl der Einsen modulo m:

$$\begin{aligned} \delta([\varepsilon], 0) &= [\varepsilon], \quad \delta([\varepsilon], 1) = [1] \\ \delta([1^i], 0) &= [1^i], \quad \delta([1^i], 1) = [1^{i+1}]. \end{aligned}$$

  Man beachte, dass für jedes $a \in \{0,1\}$ stets $\delta([x], a) = [xa]$ gilt.

Wegen Satz 2.12 (b) ist unser Automat optimal, denn wir kommen mit Index(L_m) vielen Zuständen aus!

Wir bezeichnen im folgenden die Äquivalenzklasse von x gemäß $\equiv_L$ mit $[x]$.

Der Nerode-Automat N_L **für** L**:**

- Die Zustände von N_L sind die Äquivalenzklassen von $\equiv_L$.
- Anfangszustand ist $[\varepsilon]$,
- die Menge der akzeptierenden Zustände ist

$$F = \Big\{[x] \mid x \in L\Big\}$$

- das Programm ist definiert durch

$$\delta([x], a) = [xa].$$

Satz 2.13 *$L = L(N_L)$, und N_L ist minimal: Jeder Automat, der L akzeptiert, hat mindestens so viele Zustände wie N_L.*

Beweis: Unser Vorgehen ähnelt dem Vorgehen bei der Betrachtung des Äquivalenzklassenautomaten. Zuerst überprüfen wir, ob N_L wohldefiniert ist.

Die Gefahr ist, dass $[x] = [y]$ und $[xa] \neq [ya]$ gilt. In diesem Fall hängt die Definition von δ von der Wahl eines Repräsentanten ab. Wir nehmen an, dass $[x] = [y]$ gilt. Dann folgt für alle $w \in \Sigma^*$

$$xw \in L \quad \Leftrightarrow \quad yw \in L.$$

Sei $a \in \Sigma$ ein beliebiger Buchstabe. Dann gilt für alle $w' \in \Sigma^*$

$$xaw' \in L \quad \Leftrightarrow \quad yaw' \in L,$$

und dies bedeutet, dass $[xa] = [ya]$.

Im nächsten Schritt zeigen wir, dass

$$x \in L \quad \Leftrightarrow \quad [x] \in F$$

für jedes Wort $x \in \Sigma^*$ gilt. Dies folgt aus

$$\begin{aligned} [x] \in F \quad &\Leftrightarrow \quad \text{Es gibt } y \in L \text{ mit } [x] = [y]. \\ &\Leftrightarrow \quad \text{Es gibt } y \in L, \text{ so dass } x \text{ und } y \text{ zum selben (akzeptierenden)} \\ & \quad \text{Zustand führen.} \\ &\Leftrightarrow \quad x \in L. \end{aligned}$$

Wir kommen jetzt zum Nachweis von $L = L(N_L)$. Sei $x = x_1 \cdots x_n \in \Sigma^n$. Dann gilt, mit Startzustand $[\varepsilon]$,

$$\begin{aligned} \delta([\varepsilon], x_1 \cdots x_n) &= \delta(\delta([\varepsilon], x_1), x_2 \cdots x_n) \\ &= \delta([\varepsilon x_1], x_2 \cdots x_n) \\ &= \delta([x_1], x_2 \cdots x_n) \\ &= \delta([x_1 x_2], x_3 \cdots x_n) \\ &= \delta([x_1 \cdots x_{n-1}], x_n) \\ &= [x_1 \cdots x_n] \\ &= [x] \end{aligned}$$

und wir erhalten

$$\begin{aligned} N_L \text{ akzeptiert } x \quad &\Leftrightarrow \quad [x] \in F \\ &\Leftrightarrow \quad x \in L. \end{aligned}$$

Die Minimalität von N_L ist eine direkte Konsequenz von Satz 2.12 (2). □

Wir bringen als nächstes die Ernte ein:

Satz 2.14 *Der Satz von Myhill und Nerode.*

(a) *Sei L regulär. Dann stimmt die Anzahl der Zustände des minimalen endlichen Automaten für L mit dem Index von L überein.*

(b) L ist genau dann regulär, wenn der Index von L endlich ist.

Beweis (a): Nach Satz 2.13 ist der Nerode-Automat N_L minimal. Die Behauptung folgt, da die Anzahl der Zustände von N_L mit dem Index von L übereinstimmt.

(b) „$\Rightarrow$" Wenn L regulär ist, dann gibt es einen endlichen Automaten A mit $L = L(A)$. Nach Satz 2.12 gilt

$$\text{Index}(L) \leq |Q| < \infty$$

„$\Leftarrow$" Wenn der Index von L endlich ist, dann hat N_L endlich viele Zustände. N_L ist also ein endlicher Automat. Da $L = L(N_L)$, muß L eine reguläre Sprache sein. □

Als eine erste Konsequenz des Satzes von Nerode können wir Aussagen über nicht reguläre Sprachen machen, da L genau dann icht regulär ist, wenn der Index von L unendlich ist.

Beispiel 2.11 Die Sprache $L = \{a^n\, b^n \mid n \in \mathbb{N}\}$ ist nicht regulär, da der Index von L unendlich ist: Es ist $a^n \not\equiv_L a^m$ für $n \neq m$, da $a^n\, b^n \in L$, aber $a^m\, b^n \notin L$.

Natürlich k2onnen wir in diesem Fall auch das Pumping-Lemma anwenden. Man beachte aber, dass das Pumping-Lemma für nicht-reguläre Sprachen versagen kann, der Index der Sprache hingegen „spricht immer die Wahrheit".

Die Minimalität des Äquivalenzklassenautomaten ist eine zweite Konsequenz.

Satz 2.15 *Der endliche Automat A sei vorgegeben, wobei A keine überflüssigen Zustände besitzt. Dann ist der Äquivalenzklassenautomat A' minimal.*

Beweis: Sei $A' = (Q, \Sigma, \delta, q_0, F)$ der Äquivalenzklassenautomat von A. Wir werden zeigen, dass

$$x \equiv_L y \quad \Leftrightarrow \quad \delta(q_0, x) = \delta(q_0, y) \tag{2.5}$$

gilt. Mit anderen Worten, die Menge aller Eingaben, die einen fixierten Zustand q im Äquivalenzklassenautomaten erreichen, stimmt mit einer Äquivalenzklasse der Nerode-Relation überein. Damit stimmt aber auch die Zustandszahl des Äquivalenzklassenautomaten mit dem Index von L überein und wir haben die Minimalität von A' verifiziert.

Wir verifizieren (2.5):

$$\begin{aligned} x \equiv_L y \quad &\Leftrightarrow \quad \forall w \in \Sigma^* \Big(xw \in L \quad \Leftrightarrow \quad yw \in L \Big) \\ &\Leftrightarrow \quad \forall w \in \Sigma^* \Big(\delta(q_0, xw) \in F \quad \Leftrightarrow \quad \delta(q_0, yw) \in F \Big) \\ &\Leftrightarrow \quad \forall w \in \Sigma^* \Big(\delta(\delta(q_0, x), w) \in F \quad \Leftrightarrow \quad \delta(\delta(q_0, y), w) \in F \Big) \\ &\Leftrightarrow \quad \delta(q_0, x) \equiv_{A'} \delta(q_0, y). \end{aligned}$$

Da A' der Äquivalenzklassenautomat ist, sind seine Äquivalenzklassen einelementig, und wir erhalten

$$x \equiv_L y \quad \Leftrightarrow \quad \delta(q_0, x) = \delta(q_0, y).$$

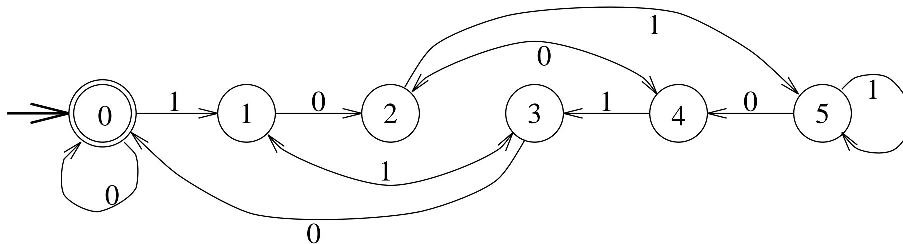
□

Bemerkung 2.2 Der Beweis hat sogar mehr gezeigt. Die Äquivalenz (2.5) besagt nämlich, dass Äquivalenzklassenautomat und Nerode-Automat isomorph sind, d.h. dass beide Automaten bis auf Umbenennung der Zustände übereinstimmen. Bis auf Umbenennung der Zustände gibt es also genau einen minimalen Automaten.

Wir können das Minimierungsproblem also wie folgt lösen. Der endliche Automat A sei vorgegeben.

1. Entferne alle überflüssigen Zustände.
2. Konstruiere den Äquivalenzklassenautomaten.

Beispiel 2.12 Der Automat A mit Zustandsdiagramm



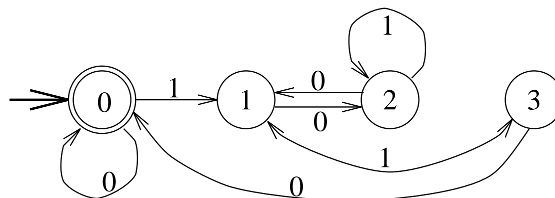
akzeptiert die Sprache

$$L = \left\{ w \in \{0,1\}^* \;\middle|\; \sum_{i=1}^{|w|} w_i \, 2^{|w|-i} \equiv 0 \ (6) \right\},$$

also die Menge der Binärdarstellungen von Zahlen, die kongruent 0 modulo 6 sind.

Als erstes überprüfen wir mittels Tiefensuche vom Anfangszustand ⓪ aus, dass A keine überflüssigen Zustände besitzt. Sodann berechnen wir A'. Zuerst erkennen wir, dass Zustand ⓪ der einzige akzeptierende Zustand und damit nicht äquivalent zu einem anderen Zustand ist. Wir können also alle Paare {⓪,①}, {⓪,②}, {⓪,③}, {⓪,④} und {⓪,⑤} markieren. Unser Markierungsalgorithmus stellt fest, dass ⓪ $= \delta($③$, 0)$ und markiert die Paare {③,①}, {③,②}, {③,④} und {③,⑤} (denn zum Beispiel $\delta($①$, 0) =$ ② und ⓪ $\not\equiv_A$ ②). Damit bildet auch Zustand ③ seine eigene Äquivalenzklasse.

Zustand ③ wird von den Zuständen ④ und ① unter dem Buchstaben 1 erreicht, und wir markieren die Paare {①,②}, {④,②}, {①,⑤} und {④,⑤}. Weitere Paare können nicht markiert werden, und somit erhalten wir, dass nur die Zustände ① und ④ sowie ② und ⑤ äquivalent sind. Der minimale Automat wird somit durch das Zustandsdiagramm



gegeben.

Beispiel 2.13 Das Pattern Matching Problem.

Das Pattern-Matching Problem ist zentral für den Entwurf von Editoren. Für ein gegebenes Pattern $P \in \Sigma^*$ ist zu entscheiden, ob und wo P als Teilwort einer Datei $T \in \Sigma^*$ vorkommt. Wir fixieren P und betrachten die Sprache

$$L_P = \{T \in \Sigma^* \mid P \text{ ist ein Suffix von } T\ \}.$$

Der Sprache L_P entspricht der reguläre Ausdruck $\Sigma^* \circ \{P\}$. Da wir später zeigen, dass eine Sprache mit regulärem Ausdruck selbst regulär ist, gibt es somit einen endlichen Automaten A_P, der L_P akzeptiert. Also können wir das Pattern-Matching Problem lösen, indem wir den Automaten A_P auf die Eingabe T ansetzen und genau die Präfixe von T feststellen, für die sich A_p in einem akzeptierenden Zustand befindet.

Wie wird der Automat A_P aussehen? Das Pattern $P = P_1 \cdots P_n$ habe n Buchstaben. Dann verwenden wir als Zustandsmenge die Menge

$$\{(0, \varepsilon), (1, P_1), (2, P_2), \ldots, (n, P_n)\},$$

wobei $(0, \varepsilon)$ der Anfangszustand und (n, P_n) der einzige akzeptierende Zustand ist. Zustandsübergänge sind so einzusetzen, dass

> eine Eingabe T genau dann den Zustand (i, P_i) als letzten Zustand erreicht, wenn $P_1 \cdots P_i$ der längste Präfix von P ist, der auch Suffix von T ist.

Wenn eine Eingabe zum Beispiel den Zustand (i, P_i) erreicht hat und wenn $Q = P_{i+1}$ der nächste Buchstabe der Eingabe ist, dann ist zum Zustand $(i + 1, P_{i+1})$ zu wechseln. Ist der nächste Buchstabe Q hingegen nicht P_{i+1}, dann müssen wir den längsten Präfix von P bestimmen, der auch Suffix von $P_1 \cdots P_i Q$ ist. Endet der längste Präfix in Position j, ist ein Zustandsübergang von (i, P_i) nach (j, P_j) mit Buchstaben Q einzusetzen.

Wenn A_P bekannt ist, dann kann das Pattern-Matching Problem in Linearzeit $O(|T|)$ gelöst werden! Schneller geht's nimmer. Tatsächlich ist A_P auch ein minimaler Automat:

Aufgabe 26

Es sei Σ ein endliches Alphabet und P ein Wort aus Σ^*. Wir definieren die „Suffix-Sprache“ $L_P = \{T \in \{a, b\}^* \mid P \text{ ist ein Suffix von } T\}$. Beschreibe die Äquivalenzklassen der Nerode Relation von L_P. Zeige, dass stets

$$\text{Index}(L_P) \geq |P| + 1$$

gilt.

(Also ist der Automat A_P minimal, da A_P genau $|P| + 1$ Zustände besitzt. Insbesondere gilt also $\text{Index}(L_P) = |P| + 1$.)

Jetzt können wir auch die $|P|+1$ Nerode-Äquivalenzklassen bestimmen: Für jedes i $(0 \leq i \leq |P|)$ besteht die Nerode-Klasse $P_1 \cdots P_i$ aus allen Texten T, für die $P_1 \cdots P_i$ der längste Präfix von P ist, der auch Suffix von T ist.

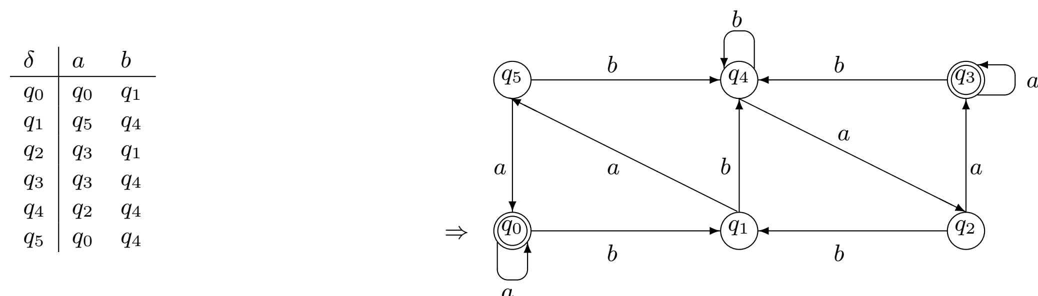
Aufgabe 27
Ein *Mealy-Automat* ist ein endlicher Automat A, der für jeden Zustandsübergang eine Ausgabe ausgibt. D.h. A ist durch den Vektor $A = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ gegeben, wobei $\lambda : Q \times \Sigma \to \Gamma$ jedem Zustandsübergang einen Buchstaben des Ausgabealphabets Γ zuweist. Als einziger Unterschied zu endlichen Automaten werden also Buchstaben aus Γ pro Zustandsübergang ausgegeben; die Komponenten Q, Σ, δ und q_0 behalten ihre ursprüngliche Bedeutung. Die Ausgabe von A als Antwort auf die Eingabe $a_1 a_2 \cdots a_n$ ist deshalb

$$\lambda(q_0, a_1)\lambda(q_1, a_2) \cdots \lambda(q_{n-1}, a_n),$$

wobei $q_0, q_1, \ldots, q_{n-1}$ die Folge von Zuständen mit $\delta(q_{i-1}, a_i) = q_i$ für $1 \leq i \leq n$ ist.

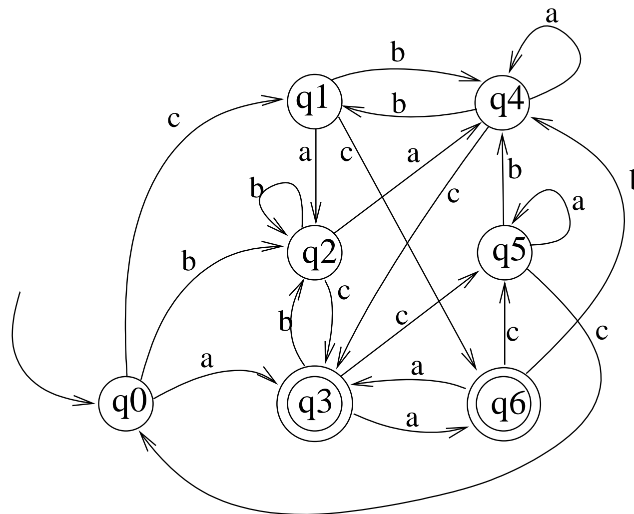
Frage: Wie ist der Äquivalenzklassenautomat für A zu definieren?

Aufgabe 28
Minimiere den folgenden endlichen Automaten. (Das Eingabealphabet ist $\{a, b\}$, die Zustandsmenge ist $\{q_0, \ldots, q_5\}$, der Anfangszustand ist q_0 und $F = \{q_0, q_3\}$ ist die Menge der akzeptierenden Endzustände.)



δ	a	b
q_0	q_0	q_1
q_1	q_5	q_4
q_2	q_3	q_1
q_3	q_3	q_4
q_4	q_2	q_4
q_5	q_0	q_4

Aufgabe 29
Minimiere den folgenden Automaten:



Aufgabe 30
Gib die Äquivalenzklassen der folgenden Sprachen über dem Alphabet $\Sigma = \{0, 1\}$ bzgl. der Nerode-Relation an:

$L_1 = \{w \in \Sigma^* \mid w$ besitzt mindestens $\left\lceil \frac{|w|}{2} \right\rceil$ Einsen $\}$,

$L_2 = \{w \in \Sigma^* \mid 4$ teilt $Bin(w)\}$. (Für eine Eingabe $w = w_1, \ldots, w_n$ bezeichne $Bin(w)$ die durch w dargestellte Binärzahl, d.h. $Bin(w) = \sum_{i=1}^{n} w_i 2^{n-i}$, sowie $Bin(\varepsilon) = 0$.)

Aufgabe 31
Seien L, L_1 und L_2 Sprachen über dem Alphabet Σ. **Beweise** oder **widerlege**:

(a) $\text{Index}(L \circ \Sigma^*) \leq \text{Index}(L)$,

(b) $\text{Index}(L_1 \cap L_2) \leq \text{Index}(L_1) \cdot \text{Index}(L_2)$,

(c) $\text{Index}(L_1 \circ L_2) \leq \text{Index}(L_1) \cdot \text{Index}(L_2)$.

Aufgabe 32
Bestimme zu gegebenem n zwei Sprachen L_1 und L_2, so dass die folgenden drei Bedingungen gelten:

1. $\text{Index}(L_1) \geq n$,
2. $\text{Index}(L_2) \geq n$ und
3. $\text{Index}(L_1 \cap L_2) = \text{Index}(L_1) \cdot \text{Index}(L_2)$

Fazit: Die Zustandszahl wächst also im schlimmsten Fall unter Durchschnittsbildung multiplikativ an.

Aufgabe 33
Sei $\Sigma = \{(,)\}$ das Alphabet aus öffnender und schließender Klammer. Die Sprache K der legalen Klammerausdrücke ist die kleinste Sprache mit den folgenden Eigenschaften.

1. $\varepsilon \in K$
2. $w \in K \rightarrow (w) \in K$
3. $u \in K \wedge v \in K \rightarrow u \circ v \in K$

Zeige, dass die Sprache K aller legalen Klammerausdrücke unendlichen Index hat.

Fazit: Damit ist also K keine reguläre Sprache.

Aufgabe 34
Wir betrachten die Sprache $L = \{0^{(k^2)} \mid k \geq 1$ ist eine natürliche Zahl $\}$.

Zeige, dass die Nerode-Relation von L unendlichen Index hat (und die Sprache L somit nicht regulär ist).

Aufgabe 35
Gegeben seien die beiden Sprachen: $L_1 = \{w \in \{a,b\}^* \mid ba$ ist Teilwort von $w\}$ und $L_2 = \{w \in \{a,b\}^* \mid ab$ ist Teilwort von $w\}$.

Konstruiere einen *minimalen* deterministischen endlichen Automaten, der die Sprache $L = L_1 \circ L_2$ erkennt.

2.4 Abschlusseigenschaften und Entscheidungsprobleme

Manchmal erhalten wir eine noch kürzere Beschreibung, wenn wir ε-Übergänge zulassen:

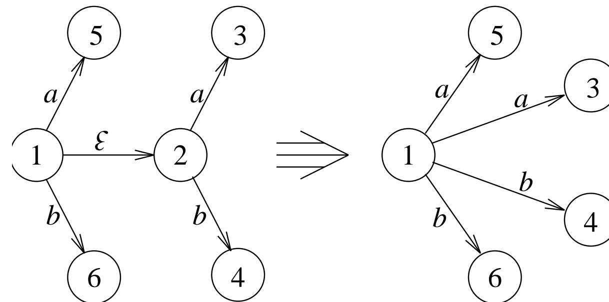
Definition 2.16

Ein nichtdeterministischer Automat $(Q, \Sigma, \delta, q_0, F)$ mit ε-Übergängen ist wie ein nichtdeterministischer Automat definiert. Als einziger Unterschied hat das Programm jetzt die Form

$$\delta : Q \times \Big(\Sigma \cup \{\varepsilon\}\Big) \to \mathcal{P}(Q)$$

Der Automat darf also ε-Übergänge benutzen, ohne Buchstaben zu lesen.

Auch bei ε-Übergängen werden nur reguläre Sprachen akzeptiert. Dies folgt, da wir ε-Übergänge nach dem folgenden Schema entfernen können:



ε-Übergänge helfen zum Beispiel im Nachweis der folgenden Abschlusseigenschaften.

Satz 2.17 *Die Sprachen L, L_1 und L_2 seien regulär. Es folgt*

(a) $\overline{L}$ *ist regulär.*

(b) $L_1 \cup L_2$ *und* $L_1 \cap L_2$ *sind regulär.*

(c) $L_1 \circ L_2$ *ist regulär.*

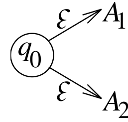
(d) L^* *ist regulär.*

Beweis: Die deterministischen Automaten A, A_1 bzw. A_2 mögen die Sprachen L, L_1 bzw. L_2 akzeptieren.

(a) Q sei die Zustandsmenge von A. Ersetze die Menge F der akzeptierenden Zustände von A durch $Q \setminus F$.

(b) Wir zeigen nur, dass $L_1 \cup L_2$ regulär ist. (Die Regularität von $L_1 \cap L_2$ folgt mit (a), da $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.)

$L_1 \cup L_2$ ist regulär, da der Automat



genau die Vereinigung akzeptiert.

(c) Wir verbinden alle akzeptierenden Zustände von A_1 durch ε-Übergänge mit dem Startzustand von A_2. Die akzeptierenden Zustände von A_2 sind die akzeptierenden Zustände des neuen Automaten: Der neue Automat akzeptiert $L_1 \circ L_2$.

(d) Verbinde alle akzeptierenden Zustände von A durch ε-Übergänge mit einem neuen Startzustand q_0^*: Der Zustand q_0^* wird auch der einzige akzeptierende Zustand des neuen Automaten. Schließlich füge einen ε-Übergang von q_0^* zum alten Startzustand q_0 ein. Da q_0^* akzeptierend ist, wird auch das leere Wort akzeptiert, und der neue Automat akzeptiert die Sternhülle. □

Aufgabe 36
Warum ist die Konstruktion in Teil (d) falsch? Wie sollte der richtige Automat für L^* konstruiert werden?

Die Klasse der regulären Sprachen ist also unter den wichtigsten mengentheoretischen Operationen abgesclossen wie auch unter der Konkatenation und der Sternoperation.

Wir zeigen jetzt, dass viele wichtige Eigenschaften deterministischer endlicher Automaten effizient nachprüfbar sind. Entscheidungsprobleme für nichtdeterministische Automaten sind, mit einigen wenigen Ausnahmen, deutlich schwieriger.

Satz 2.18 *A, A_1 und A_2 bezeichne deterministische Automaten und N bezeichne einen nichtdeterministischen Automaten. Die folgenden Entscheidungsprobleme sind effizient lösbar:*

(a) Ist $L(A) \neq \emptyset$, bzw. ist $L(N) \neq \emptyset$?

(b) Ist $L(A) = \Sigma^*$?

(c) Ist $L(A_1) = L(A_2)$, bzw. ist $L(A_1) \subseteq L(A_2)$?

(d) Ist $L(A)$ endlich?

Beweis (a): Wir untersuchen die Frage nach Leerheit nur für einen nichtdeterministischen Automaten N, das Leerheitsproblem für deterministische Automaten kann völlig analog gelöst werden. Sei q_0 der Anfangszustand von N und sei F die Menge der akzeptierenden Zustände. Wir beobachten, dass $L(N)$ genau dann nichtleer ist, wenn es einen Weg von q_0 zu einem Zustand in F gibt. Das Erreichbarkeitsproblem für mindestens einen Zustand in F können wir jetzt mit Hilfe der Tiefensuche, angewandt auf das Zustandsdiagramm von N, effizient lösen.

(b),(c) und **(d)** sind als Übungsaufgaben gestellt. □

Aufgabe 37
Sei A ein deterministischer endlicher Automat. Entwerfe einen möglichst effizienten Algorithmus, der entscheidet ob A unendlich viele Worte akzeptiert. Bestimme die Laufzeit in Abhängigkeit von der Anzahl der Zustände von A.

Bemerkung 2.3 Die Entscheidungsprobleme (b), (c) und (d) aus Satz 2.18 lassen sich in aller Wahrscheinlichkeit nicht effizient für nichtdeterministische Automaten lösen. Erstaunlicherweise gehört das anscheinend unschuldige Problem $L(N) \stackrel{?}{=} \Sigma^*$ noch nicht einmal zur Klasse `NP`: Einer der Gründe hierfür ist, dass sich der Unterschied zwischen $L(N)$ und Σ^* erst für Worte zeigen kann, deren Länge exponentiell in der Anzahl der Zustände ist! Auch das Minimierungsproblem für nichtdeterministische Automaten ist anscheinend äußerst schwierig.

Viele vernünftige Entscheidungsprobleme lassen sich effizient für deterministische Automaten lösen. Eine Ausnahme ist das Interpolationsproblem: Für gegebene Teilmengen $P, N \subseteq \Sigma^*$ und einen Schwellenwert N, gibt es einen deterministischen endlichen Automaten mit höchstens N Zuständen, der alle Worte in P akzeptiert und alle Worte in N verwirft?

2.5 Reguläre Ausdrücke

Wir führen ein weiteres Beschreibungsschema regulärer Sprachen ein, nämlich reguläre Ausdrücke. Wir haben reguläre Ausdrücke bereits verwendet, zum Beispiel bei der Definition der Sprache $L_k = \Sigma^* \circ \{1\} \circ \Sigma^k$: Diese Sprache besitzt NFA's mit wenigen Zuständen wie auch einen kurzen regulären Ausdruck, aber nur deterministische endliche Automaten mit exponentiell in k vielen Zuständen.

Definition 2.19 *Das Alphabet $\Sigma = \{a_1, \ldots, a_k\}$ sei gegeben.*

- *Wir geben eine Definition regulärer Ausdrücke an:*

  (a) $\emptyset$, ε, $a_1, \ldots, a_k$ sind reguläre Ausdrücke (für die Sprachen $\emptyset$, $\{\varepsilon\}$ und $\{a_1\}, \ldots, \{a_k\}$.)

  (b) Sei R ein regulärer Ausdruck (für die Sprache L), dann ist R^* ein regulärer Ausdruck (für die Sprache L^*), und (R) ist ein regulärer Ausdruck (für die Sprache L).

  (c) Seien R_1 und R_2 reguläre Ausdrücke (für die Sprachen L_1 und L_2). Dann sind auch

  $$R_1 + R_2 \quad \textit{und} \quad R_1 \cdot R_2$$

  reguläre Ausdrücke (für die Sprachen $L_1 \cup L_2$ und $L_1 \circ L_2$).

  Die Menge der regulären Ausdrücke ist die kleinste Menge mit den Eigenschaften (a), (b) und (c).

- *Wenn R ein regulärer Ausdruck für eine Sprache L ist, dann sagen wir auch, dass R die Sprache L beschreibt und definieren*

  $$L(R) = L.$$

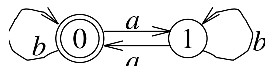
Beispiel 2.14 Reguläre Ausdrücke für die Sprachen

$$\begin{aligned} L_1 &= \Big\{ w \in \{a,b\}^* \,|\, w \text{ beginnt und endet mit } a \Big\} \\ L_2 &= \Big\{ w \in \{a,b\}^* \,|\, aba \text{ ist Teilwort von } w \Big\} \\ L_3 &= \Big\{ w \in \{a,b\}^* \,|\, w \text{ hat eine gerade Anzahl von } a\text{'s} \Big\} \end{aligned}$$

sind zu konstruieren. Wir erhalten zum Beispiel

$$\begin{aligned} R_1 &= a \cdot (a+b)^* \cdot a \\ R_2 &= (a+b)^* \cdot a \cdot b \cdot a \cdot (a+b)^* \\ R_3 &= b^* + (b^* \cdot a \cdot b^* \cdot a \cdot b^*)^*. \end{aligned}$$

Während die regulären Ausdrücke für L_1 und L_2 einfach sind (und sich der Formalismus regulärer Ausdrücke auszahlt), ist der reguläre Ausdruck für L_3, verglichen mit dem Automaten



relativ kompliziert: Jeder Formalismus hat seine Stärken und Schwächen.

Satz 2.20 *Wenn R ein regulärer Ausdruck ist, dann ist $L(R)$ regulär.*

Beweis: Offensichtlich sind

$$\emptyset, \{\varepsilon\}, \{a_1\}, \ldots, \{a_k\}$$

reguläre Sprachen. Mit Satz 2.17 sind aber auch L^*, $L_1 \circ L_2$ und $L_1 \cup L_2$ regulär, wenn L, L_1 und L_2 regulär sind. Deshalb führen auch die regulären Ausdrücke

$$R^*, \quad R_1 \cdot R_2 \quad \text{und} \quad R_1 + R_2$$

auf reguläre Sprachen (wenn wir induktiv annehmen, dass R, R_1 und R_2 auf reguläre Sprachen führen). □

In den beiden nächsten Sätzen sehen wir, dass <u>reguläre Ausdrücke und reguläre Sprachen identische Konzepte sind</u>:

Satz 2.21 *Sei L regulär über Σ. Dann gibt es einen regulären Ausdruck für L.*

Beweis: Da L regulär ist, gibt es einen (deterministischen) endlichen Automaten A, der L akzeptiert. Ohne Beschränkung der Allgemeinheit sei

$$Q = \{1, \ldots, n\}$$

die Zustandsmenge von A und $q_0 = 1$ sein Anfangszustand. Wir betrachten die Menge

$$L^k_{p,q} = \Big\{ w \in \Sigma^* \quad | \quad \delta(p,w) = q, \text{ und alle } \underline{\text{Zwischen}}\text{zustände während des Lesens von } w \text{ liegen in der Menge } \{1, \ldots, k\} \Big\}$$

Wir beachten, dass

$$L = \bigcup_{p \in F} L^n_{1,p}$$

und es genügt deshalb zu zeigen, dass jede Sprache $L^n_{p,q}$ als regulärer Ausdruck darstellbar ist. Wir zeigen mehr, nämlich dass jede Sprache $L^k_{p,q}$ durch einen regulären Ausdruck darstellbar ist. Unsere Konstruktion verläuft rekursiv (nach k).

k = 0: $L^0_{p,q}$ ist die Menge aller $w \in \Sigma^*$ mit $\delta(p,w) = q$, wobei **kein** Zwischenzustand erlaubt ist. Also ist

$$L^0_{p,q} = \Big\{a \in \Sigma \,|\, \delta(p,a) = q\Big\}, \qquad \text{falls } p \neq q, \text{ bzw.}$$

$$L^0_{p,q} = \{\varepsilon\} \cup \Big\{a \in \Sigma \,|\, \delta(p,a) = p\Big\}, \quad \text{falls } p = q.$$

Die endliche Menge $L^0_{p,q}$ ist offensichtlich durch einen regulären Ausdruck darstellbar.

k → k + 1: Es ist

$$L^{k+1}_{p,q} = L^k_{p,q} \cup L^k_{p,k+1} \circ \left(L^k_{k+1,k+1}\right)^* \circ L^k_{k+1,q},$$

denn entweder taucht Zustand $k+1$ beim Lesen von w (mit Startzustand p und Endzustand q) nicht auf, oder wir können die Folge der Zwischenzustände so zerlegen, dass in jeder Teilfolge $k+1$ kein Zwischenzustand, sondern nur Endzustand ist. Wenn aber die Sprachen $L^k_{p,q}$, $L^k_{p,k+1}$, $L^k_{k+1,k+1}$ und $L^k_{k+1,q}$ durch reguläre Ausdrücke beschrieben werden können, dann kann auch $L^{k+1}_{p,q}$ durch einen regulären Ausdruck beschrieben werden. (Beachte, dass wir einen Beweis mit Hilfe der dynamischen Programmierung geführt haben.) □

Aufgabe 38
Wir definieren die *Länge* eines regulären Ausdrucks als die Anzahl der Symbole des Ausdrucks, wobei die Klammern nicht gezählt werden.

Beweise oder **widerlege** die folgende Aussage: Für alle regulären Sprachen L gilt: Gegeben sei ein regulärer Ausdruck R der Länge l (für die Sprache L). Dann gibt es einen nichtdeterministischen endlichen Automaten mit $O(l)$ Zuständen, der die Sprache L erkennt.

Aufgabe 39
Gib einen möglichst kurzen regulären Ausdruck für die folgenden Sprachen über $\Sigma = \{0,1\}$ an:

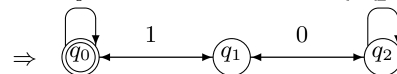
(a) $L = \{w \in \{0,1\}^* \mid w$ beginnt mit 1 und enthält das Teilwort 00 nicht $\}$,

(b) $L_k = \{w \in \{0,1\}^* \mid$ die Anzahl der Einsen in w ist durch k teilbar $\}$.

Aufgabe 40
Wir führen in dieser Aufgabe die erweiterten regulären Ausdrücke ein: Jeder reguläre Ausdruck ist auch ein erweiterter regulärer Ausdruck. In erweiterten regulären Ausdrücken lassen wir zusätzlich die Operation $\cap$ zu, d.h. wenn R_1 und R_2 erweiterte reguläre Ausdrücke für die Sprachen L_1 und L_2 sind, dann ist $R_1 \cap R_2$ ein erweiterter regulärer Ausdrück für die Sprache $L_1 \cap L_2$.

Beweise oder **widerlege** die folgende Aussage: Für alle regulären Sprachen L gilt: Gegeben sei ein erweiterter regulärer Ausdruck R der Länge l (für L). Dann gibt es einen nichtdeterministischen endlichen Automaten mit $O(poly(l))$ Zuständen (der L erkennt).

Aufgabe 41
Gegeben sei der folgende deterministische endliche Automat A über dem Eingabealphabet $\Sigma = \{0,1\}$ mit der Zustandsmenge $Q = \{q_0, q_1, q_2\}$, dem Anfangszustand q_0 und $F = \{q_0\}$:



Bestimme gemäß der rekursiven Konstruktion aus dem Beweis von Satz 2.21 einen regulären Ausdruck, der die Sprache $L(A)$ darstellt.

Aufgabe 42
Es seien r und s reguläre Ausdrücke über disjunkten Alphabeten, und es sei das leere Wort $\varepsilon \notin r$. **Bestimme** einen regulären Ausdruck x, der die Gleichung $x = r \circ x + s$ erfüllt. Dabei bezeichnet $\circ$ die Konkatenation und $+$ die Vereinigung.

2.6 Grammatiken und reguläre Grammatiken

Was ist eine Grammatik?

Definition 2.22

(a) *Eine Grammatik G hat die folgenden Komponenten:*

- *ein endliches Alphabet Σ,*
- *eine endliche Menge V von Variablen (oder Nichtterminalen) mit $\Sigma \cap V = \emptyset$,*
- *das Startsymbol $S \in V$ und*
- *eine endliche Menge P von Produktionen, wobei eine Produktion von der Form (u, v) ist mit*

$$u \in (\Sigma \cup V)^* V (\Sigma \cup V)^* \quad \textit{und} \quad v \in (\Sigma \cup V)^*$$

(b) *Sei (u, v) eine Produktion von G. Dann definieren wir für Worte w_1 und $w_2 \in (\Sigma \cup V)^*$*

$$w_1 \rightarrow w_2 \quad \Leftrightarrow \quad \textit{Es gibt } x, y \in (\Sigma \cup V)^* \textit{ mit } w_1 = xuy \textit{ und } w_2 = xvy.$$

(c) *Seien $r, s \in (\Sigma \cup V)^*$. Dann definieren wir*

$$r \stackrel{*}{\rightarrow} s \quad \Leftrightarrow \quad \textit{Es gibt Worte } w_1 = r, w_2, \ldots, w_k = s, \textit{ so dass } w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_k.$$

(d) $L(G) = \left\{ w \in \Sigma^* \mid S \stackrel{*}{\rightarrow} w \right\}$ *ist die von der Grammatik erzeugte Sprache.*

Warum betrachten wir Grammatiken? Programmiersprachen lassen sich am besten als eine formale Sprache, die Sprache aller syntaktisch korrekten Programme auffassen. Eine Grammatik für eine Programmiersprache drückt dann die Regeln der Programmiersprache aus.

Beispiel 2.15 Wir geben eine Grammatik G für arithmetische Ausdrücke mit ganzzahligen Komponenten an. G besitzt das Eingabealphabet

$$\Sigma = \left\{ +, -, *, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,) \right\}$$

sowie

- das Startsymbol A und
- die Variablen I und Z.

Die Produktionen haben die Form

$$\begin{array}{rcl} A & \rightarrow & A + A \,|\, A - A \,|\, A * A \,|\, (A) \,|\, I \,|\, +I \,|\, -I \\ I & \rightarrow & ZI \,|\, Z \\ Z & \rightarrow & 0 \,|\, 1 \,|\, 2 \,|\, 3 \,|\, 4 \,|\, 5 \,|\, 6 \,|\, 7 \,|\, 8 \,|\, 9 \end{array}$$

(Die vertikalen Linien trennen Produktionen, die dieselbe Variable ersetzen.) Die Regeln dieser Grammatik spiegeln die rekursive Definition arithmetischer Ausdrücke wider.

Die erzeugte Sprache $L(G)$ ist nicht regulär. (Warum?) Wenn wir aber das Startsymbol A durch I ersetzen, erhalten wir eine reguläre Sprache, nämlich die Sprache der Dezimaldarstellungen natürlicher Zahlen.

Definition 2.23 *Eine Grammatik $G = (\Sigma, V, S, P)$ heißt* <u>*regulär*</u>*, wenn alle Produktionen in P die Form*

$$\begin{aligned} u &\to \varepsilon && (\text{für } u \in V) \text{ oder} \\ u &\to av && (\text{für } u, v \in V,\ a \in \Sigma) \end{aligned}$$

haben.

Satz 2.24 *L sei eine Sprache. Dann gilt:*

$$L \text{ regulär} \quad \Leftrightarrow \quad \text{Es gibt eine reguläre Grammatik } G \text{ mit } L = L(G).$$

Beweis: „$\Rightarrow$“ Sei L regulär und sei A ein endlicher Automat A, der L akzeptiert. Auf Eingabe w führt A die Zustandsübergänge

$$q_0 \overset{w_1}{\to} q_1 \overset{w_2}{\to} \ldots \overset{w_n}{\to} q_n$$

durch und akzeptiert, falls $q_n \in F$. Dies legt nahe, eine Grammatik zu konstruieren, die Ableitungsfolgen der Form

$$q_0 \to w_1 q_1 \to w_1 w_2 q_2 \to \ldots \to w_1 \cdots w_n q_n$$

erlaubt mit anschließender Produktion $q_n \to \varepsilon$. Deshalb definieren wir eine Grammatik G mit

- Variablenmenge Q,
- Startymbol q_0,
- sowie den Produktionen

$$\begin{aligned} p &\to aq && \text{falls } \delta(p, a) = q \\ p &\to \varepsilon && \text{falls } p \in F. \end{aligned}$$

„$\Leftarrow$“ Angenommen, $L = L(G)$ für eine reguläre Grammatik $G = (\Sigma, V, S, P)$. Eine Ableitung von G hat die Form

$$S \to w_1 V_1 \to w_1 w_2 V_2 \to \ldots \to w_1 \cdots w_n V_n \to w_1 \cdots w_n.$$

Eine Simulation durch einen deterministischen Automaten scheint nicht sinnvoll, da im allgemeinen aus mehreren möglichen Produktionen gewählt werden kann. Es bietet sich vielmehr eine Simulation durch einen nichtdeterministischen Automaten N an. Wir setzen

- $Q = V$,
- $q_0 = S$

und definieren

- $\delta(X, a) = \{Y \mid X \to aY \ \in P\}$
- $F = \{X \in V \mid X \to \varepsilon \ \in P\}$

□

Aufgabe 43
Beweise oder **widerlege**: Sei L regulär. Dann stimmt die minimale Zustandszahl eines NFA A für L überein mit der kleinsten Zahl von Variablen einer regulären Grammatik G für L.

2.7 Zusammenfassung

Wir haben endliche Automaten und die von ihnen akzeptierten Sprachen, die regulären Sprachen, betrachtet. Wir haben eine *Minimierungsprozedur* besprochen, die zu einem gegebenen deterministischen endlichen Automaten einen äquivalenten, aber minimalen Automaten, den *Äquivalenzklassenautomaten* bestimmt. Der Beweis der Minimalität gelang durch den *Index der Sprache*, der der Minimalanzahl von Zuständen entspricht. Unsere Argumentation hat auch gezeigt, dass der minimale Automat, bis auf eine Umbenennung der Zustände, eindeutig bestimmt ist.

Während der Betrachtung minimaler Automaten haben wir auch den *Satz von Nerode* erhalten, der reguläre Sprachen als Sprachen mit endlichem Index charakterisiert. Der Nachweis der Nichtregularität einer Sprache kann deshalb mit den Satz von Nerode oder mit dem *Pumping-Lemma* geführt werden.

Wir haben verschiedene äquivalente Charakterisierungen regulärer Sprachen kennengelernt, nämlich durch

- deterministische endliche Automaten,
- nichtdeterministische endliche Automaten,
- reguläre Ausdrücke oder
- reguläre Grammatiken.

Weiterhin haben wir gesehen, dass reguläre Sprachen unter den Operationen

- Vereinigung, Durchschnitt und Komplement
- sowie Konkatenation und Kleene-Abschluß (Sternoperation)

abgeschlossen sind. Den Abschluß unter vielen weiteren Operationen besprechen wir in den Übungen. Wichtige algorithmische Probleme wie die Feststellung der Äquivalenz oder die Minimierung lassen sich für deterministische endliche Automaten effizient beantworten, sind aber für nichtdeterministische endliche Automaten notorisch schwierig: Die größere Beschreibungskraft nichtdeterministischer endlicher Automaten muss algorithmisch teuer erkauft werden.

Kapitel 3

Kontextfreie Sprachen

Als Grundlage für die Syntaxdefinition von Programmiersprachen sind reguläre Sprachen zu ausdrucksschwach. Deshalb betrachten wir in diesem Kapitel die weitaus mächtigere Klasse der kontextfreien Sprachen, die wir durch kontextfreie Grammatiken einführen. Wir zeigen wie das Wortproblem für kontextfreie Sprachen mit Hilfe der Chomsky Normalform effizient gelöst werden kann und stellen Techniken vor, die es erlauben, Sprachen als nicht kontextfrei nachzuweisen. Am Ende des Kapitels werden wir mit Hilfe von Kellerautomaten auf die Klasse der deterministisch kontextfreien Sprachen geführt. Sie stellen den praxistauglichsten Kompromiss zwischen Ausdruckskraft und Komplexität des Wortproblems dar.

Definition 3.1 *Eine Grammatik G mit Produktionen der Form*

$$u \to v \quad \textit{mit } u \in V \textit{ und } v \in (V \cup \Sigma)^*$$

heißt kontextfrei. Eine Sprache L heißt kontextfrei, wenn es eine kontextfreie Grammatik G mit

$$L(G) = L$$

gibt.

Beispiel 3.1 Wir beschreiben einen allerdings sehr kleinen Ausschnitt von Pascal durch eine kontextfreie Grammatik. Dazu benutzen das Alphabet

$$\Sigma = \{\texttt{a}, \ldots, \texttt{z}, ;, :=\}$$

und die Variablen

$$\begin{aligned} V &= \{S, \text{statements, statement, assign-statement, while-statement,} \\ &\quad \text{variable, boolean, expression }\}. \end{aligned}$$

In den Produktionen unseres Pascal Fragments führen wir die Variablen „boolean, expression und variable“ nicht weiter aus:

$$\begin{aligned} S &\to \texttt{begin}\ \text{statements}\ \texttt{end} \\ \text{statements} &\to \text{statement} \,|\, \text{statement ; statements} \\ \text{statement} &\to \text{assign} - \text{statement} \,|\, \text{while} - \text{statement} \\ \text{assign} - \text{statement} &\to \text{variable} := \text{expression} \\ \text{while} - \text{statement} &\to \texttt{while}\ \text{boolean}\ \texttt{do}\ \text{statements} \end{aligned}$$

Lassen sich denn stets die syntaktisch korrekten Programme einer modernen Programmiersprache als eine kontextfreie Sprache auffassen? Die erste Antwort ist Nein. In Pascal muss zum Beispiel sichergestellt werden, dass Anzahl und Typen der formalen und aktuellen Parameter übereinstimmen.

> Die „Kopiersprache“ $\{ww \mid w \in \Sigma^*\}$ modelliert die Überprüfung auf Typkonsistenz, allerdings wird sich die Kopiersprache als nicht kontextfrei herausstellen.

Die zweite Antwort ist aber „im wesentlichen Ja, wenn man „Details“ wie Typ-Deklarationen und Typ-Überprüfungen ausklammert.

> Typischerweise beschreibt man die Syntax durch eine kontextfreie Grammatik, die alle syntaktisch korrekten Programme erzeugt. Allerdings werden auch syntaktisch inkorrekte Progamme (z.B. aufgrund von Typ-Inkonsistenzen) erzeugt. Die nicht eingehaltenene (nicht-kontextfreien) Syntax-Vorschriften können nach Erstellung des Ableitungsbaums überprüft werden.

Was ist ein Ableitungsbaum? Das klären wir im nächsten Abschnitt.

3.1 Ableitungsbäume

Im Compilerproblem möchten wir nicht nur entscheiden, ob ein Wort w von einer kontextfreien Grammatik erzeugt werden kann, sondern wir möchten zusätzlich eine Ableitung von w, bzw. einen Ableitungsbaum von w konstruieren.

Definition 3.2 *Sei $G = (\Sigma, V, S, P)$ eine kontextfreie Grammatik und sei B ein Baum mit den folgenden Eigenschaften:*

- *Die Wurzel von B ist mit S markiert,*
- *innere Knoten sind mit Variablen markiert, während Blätter mit Buchstaben aus Σ markiert sind,*
- *wenn der Knoten v mit der Variablen A markiert ist und wenn die Kinder von v die Markierungen (von links nach rechts) $v_1, \ldots, v_s$ tragen, dann ist $A \to v_1 \cdots v_s$ eine Produktion von G.*

B heißt ein Ableitungsbaum des Wortes w, wenn man w durch die links-nach-rechts Konkatenation der Markierungen der Blätter von B erhält.

Beispiel 3.2 Wir geben zuerst eine kontextfreie Grammatik für die Sprache

$$L = \left\{ w \in \{0,1\}^+ \mid w \text{ hat gleich viele Nullen wie Einsen} \right\}$$

an. Unsere Grammatik G besitzt das Startsymbol S sowie die Variablen Null und Eins. Alle von Null (bzw. Eins) ableitbaren Worte werden genau eine Null (bzw. Eins) mehr als Einsen (bzw. Nullen) besitzen. Dies wird erreicht durch die Produktionen

$$\begin{aligned} S &\to 0\,\text{Eins} \mid 1\,\text{Null} \\ \text{Eins} &\to 1 \mid 0\,\text{Eins}\,\text{Eins} \mid 1\,S \\ \text{Null} &\to 0 \mid 1\,\text{Null}\,\text{Null} \mid 0\,S \end{aligned}$$

(Zeige zuerst durch Induktion über die Länge der Ableitungen, dass

$$\Big\{w \in \{0,1\}^+ \,|\, \text{Eins} \xrightarrow{*} w\Big\} = \Big\{w \in \{0,1\}^+ \,|\, w \text{ hat eine Eins mehr}\Big\}$$

und

$$\Big\{w \in \{0,1\}^+ \,|\, \text{Null} \xrightarrow{*} w\Big\} = \Big\{w \in \{0,1\}^+ \,|\, w \text{ hat eine Null mehr}\Big\}$$

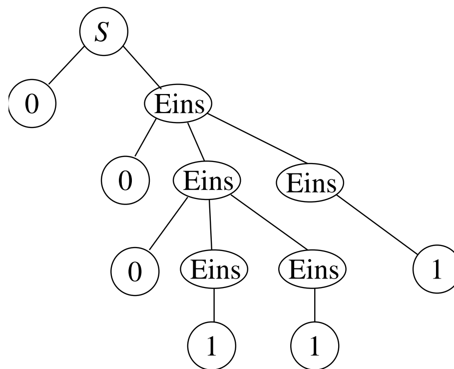
gilt. Dann folgt die Behauptung $L = L(G)$.)

Wir betrachten als Nächstes das Wort $w = 000111$. w besitzt die Ableitung

$$S \to 0\,\text{Eins} \to 00\,\text{Eins}\,\text{Eins} \to 000\,\text{Eins}\,\text{Eins}\,\text{Eins} \to$$

$$0001\,\text{Eins}\,\text{Eins} \to 00011\,\text{Eins} \to 000111.$$

und deshalb den Ableitungsbaum



Beachte, dass jedem Ableitungsbaum verschiedene Ableitungen entsprechen, nämlich zum Beispiel Linksableitungen (die jeweils die linkeste Variable ersetzen), sowie Rechtsableitungen (die jeweils die rechteste Variable ersetzen):

$$S \to 0\,\text{Eins} \to 00\,\text{Eins}\,\text{Eins} \to 00\,\text{Eins}\,1 \to$$

$$000\,\text{Eins}\,\text{Eins}\,1 \to 000\,\text{Eins}\,11 \to 000111.$$

Ordnet ein Compiler unterschiedlichen Ableitungsbäumen unterschiedliche Bedeutung zu, dann kann ein Wort, abhängig von der Ableitung unterschiedliche Bedeutungen besitzen: Ein Programm, das in einer mehrdeutigen Programmiersprache geschrieben wurde, besitzt somit compilerabhängige Semantiken! Wir sollten also schon auf eindeutigen Grammatiken bestehen!

Definition 3.3 *Sei G eine kontextfreie Grammatik.*

(a) G heißt genau dann mehrdeutig, wenn es ein Wort $w \in L(G)$ mit zwei verschiedenen Ableitungsbäumen gibt.

Wenn G nicht mehrdeutig ist, dann heißt G eindeutig.

(b) Eine Sprache L heißt eindeutig*, wenn es eine eindeutige Grammatik G gibt mit $L = L(G)$. Ansonsten heißt L inhärent mehrdeutig.*

Beispiel 3.3 Es kann gezeigt werden, dass $L = \left\{ a^i b^j c^k \mid i = j \text{ oder } j = k \right\}$ inhärent mehrdeutig ist. Woran liegt das? Wenn $w \in L$, dann gilt $w = a^i b^i c^k$ *oder* $w = a^i b^j c^j$. Eine Grammatik „sollte" deshalb eine der zwei Optionen auswählen müssen; wenn aber beide Optionen zutreffen, also wenn $w = a^i b^i c^i$, dann hat w zwei verschiedene Ableitungsbäume und die Grammatik ist mehrdeutig.

Beispiel 3.4 Die Grammatik G besitze die Komponenten

- $\Sigma = \{x, y, +, -, *, (,)\}$
- $V = \{S\}$
- die Produktionen

$$S \to S + S \mid S * S \mid (S) \mid x \mid y.$$

Damit ist $L(G)$ die Sprache aller (teilweise geklammerten) arithmetischen Ausdrücke in den Variablen x und y. G ist nicht eindeutig, denn $x + x * y$ besitzt die zwei Ableitungsbäume

Ableitungsbäume werden bei der Compilierung zur Codegenerierung benutzt. Der erste Baum liefert dann den Code $x + (x * y)$, während der zweite Baum den Code $(x + x) * y$ liefert. Es wäre also hilfreich, wenn die Sprache L eindeutig wäre, und dies ist auch tatsächlich der Fall, wenn wir eine andere Grammatik G benutzen:

Die Grammatik G mit dem Startsymbol S und den Variablen T (generiert Terme) und F (generiert Faktoren) wird eindeutig sein. Die Produktionen von G haben die Form

$$\begin{aligned} S &\to S + T \mid T \\ T &\to T * F \mid F \\ F &\to (S) \mid x \mid y. \end{aligned}$$

Mit dieser Grammatik legen wir fest, dass die Multiplikation „starker bindet" als die Addition. Wir zeigen durch Induktion über die Länge eines arithmetischen Ausdrucks A, dass G eindeutig ist.

Basis: Es ist $A \equiv x$ oder $A \equiv y$.

Zum Beispiel für $A \equiv x$ ist $S \to T \to F \to x$ die (sogar)[1] eindeutige Ableitung.

Induktion: Wir überprüfen exemplarisch den Fall $A \equiv A_1 + A_2$.

$+$ kann nur durch die Produktion $S \to S + T$ eingeführt werden. Also wird eine Ableitung von A mit der Produktion $S \to S + T$ beginnen. Die von S und T abgeleiteten Ausdrücke sind aber kürzer, und somit sind ihre Ableitungsbäume eindeutig. Also besitzt A genau einen Ableitungsbaum. □

[1] Beachte, dass eindeutige Grammatiken verschiedene Ableitungen für dasselbe Wort w besitzen können, nämlich zum Beispiel die Links- und die Rechtsableitung von w. Alle Ableitungen von w müssen aber zum eindeutig bestimmten Ableitungsbaum von w gehören.

Aufgabe 44
Beschreibe kontextfreie Grammatiken für die Sprachen L_1 und L_2:

$L_1 = \{a^i b^j c^k \mid i,\, j,\, k \geq 0 \text{ und } i \neq j + k\}$.

Es sei $\Sigma = \{x,\ y,\ (,\ ),\ +,\ *\}$. L_2 ist die Menge aller arithmetischen Ausdrücke über Σ (mit Variablen x und y, sowie mit den Operationen Addition und Multiplikation). Ein arithmetischer Ausdruck darf aber **keine** überflüssigen Klammern enthalten. Wir folgen den konventionellen Prioritätsregeln, nämlich dass die Multiplikation eine höhere Priorität hat als die Addition.

Zum Beispiel gehören die Ausdrücke $x * x * y$, $(x * y + x) * y$, $x + y + x$ zu L_3, während die Ausdrücke $(x * x) * y$, $((x * y) + x) * y$, $(x + y) + x$ überflüssige Klammern besitzen und deshalb nicht zu L_3 gehören.

Aufgabe 45
Es sei $L = \{w \in \{0,1\}^* | \text{für jedes Präfix } u \text{ von } w \text{ gilt: } |u|_0 \geq |u|_1\}$. ($|u|_b$ zählt die Anzahl der Vorkommen von b in u.)

Konstruiere eine kontextfreie Grammatik G, die die Sprache L erzeugt. **Beweise**, dass $L(G) = L$ ist.

Aufgabe 46
Sei $L = \{a^n b^m c^{n+m} :\ n, m \geq 0\}$.

(a) **Zeige** mit dem Pumping-Lemma, dass L nicht regulär ist.

(b) **Zeige**, dass L kontextfrei ist.

Aufgabe 47
Es sei $L = \{a^i b^j c^k \mid k \neq i + j\}$. **Konstruiere** eine eindeutige kontextfreie Grammatik für die Sprache L. **Beweise** die Eindeutigkeit.

Aufgabe 48
Die Grammatik G hat die folgenden Komponenten: Das Alphabet $\Sigma = \{\text{if, then, else, } a,\ b\ \}$, die Variable S, das Startsymbol S und die Produktionen $S \to \text{if } b \text{ then } S \mid \text{if } b \text{ then } S \text{ else } S \mid a$.

Zeige, dass G mehrdeutig ist. Insbesondere, **finde** ein Wort $w \in L(G)$, das zwei verschiedene Ableitungsbäume besizt.

Kommentar: Buchstabe b steht (generisch) für Bedingung, Buchstabe a steht (generisch) für Anweisung.

Aufgabe 49
Sei L eine reguläre Sprache. **Zeige**, dass L eine eindeutige reguläre Grammatik besitzt.

Aufgabe 50
Zeige, dass die folgende Grammatik G *mehrdeutig* ist. G hat die folgenden Komponenten: das Alphabet $\Sigma = \{a\}$, die Variablen S und A, das Startsymbol S und die Produktionen $S \to AA$, $A \to aSa \mid a$.

Beschreibe eine äquivalente, aber eindeutige kontextfreie Grammatik G'. (Wir nennen G und G' äquivalent, wenn $L(G) = L(G')$.)

Hinweis: Versuche zuerst, $L(G)$ zu bestimmen.

Aufgabe 51
Gegeben sei die folgende kontextfreie Grammatik $G = (\Sigma, V, S, P)$, mit $\Sigma = \{a, b, c, d\}$, $V = \{S, B\}$ und den Produktionen $S \Rightarrow aS \mid SB \mid d$, $B \Rightarrow Bb \mid c$.

Zeige, dass G nicht eindeutig ist.

Zeige, dass die Sprache $L(G)$ eindeutig ist.

3.2 Die Chomsky-Normalform und das Wortproblem

Wir versuchen jetzt, eine effiziente Lösung des Compilerproblems zu erhalten. Um dies vorzubereiten, werden wir zuerst zeigen, dass sich jede kontextfreie Grammatik (die nicht das leere Wort erzeugt) in Chomsky-Normalform überführen läßt.

Definition 3.4 *Eine Grammatik ist in Chomsky-Normalform, wenn alle Produktionen die Form*

$$A \to BC \quad \textit{oder} \quad A \to a$$

besitzen (für $A, B, C \in V$ und $a \in \Sigma$).

Satz 3.5 *Sei L kontextfrei mit $\varepsilon \notin L$. Dann gibt es eine Grammatik G in Chomsky-Normalform mit $L = L(G)$.*

Beweis: Es gelte $L = L(G)$ für eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$.

Schritt 1: Wir erzwingen, dass die rechte Seite einer Produktion entweder nur aus Variablen besteht oder nur aus einem Buchstaben.

Für jeden Buchstaben $a \in \Sigma$ füge die neue Variable X_a hinzu und erlaube $X_a \to a$ als neue Produktion. Jetzt kann jede Produktion

$$A \to \alpha_1 a_1 \alpha_2 a_2 \cdots \alpha_r a_r \alpha_{r+1}$$

(mit Buchstaben $a_1, \ldots, a_r$ und $\alpha_1, \ldots, \alpha_{r+1} \in V^*$) durch die Produktion

$$A \to \alpha_1 X_{a_1} \alpha_2 X_{a_2} \cdots \alpha_r X_{a_r} \alpha_{r+1}$$

ersetzt werden. P_1 sei die Menge der neuen Produktionen.

Schritt 2: Erzwinge, dass alle Produktionen eine rechte Seite der Länge höchstens zwei besitzen.

Ersetze jede Produktion

$$p \equiv A \to C_1 \cdots C_s \quad (\text{mit } s \geq 3)$$

durch die Produktionen

$$\begin{array}{rcl} A & \to & C_1 X_1^p, \\ X_1^p & \to & C_2 X_2^p, \\ & \vdots & \\ X_{s-2}^p & \to & C_{s-1} C_s \end{array}$$

P_2 sei die neue Produktionenmenge.

Schritt 3: Entferne alle ε-Produktionen $A \to \varepsilon$.

Die Produktion $A \to \varepsilon$ gehöre zu P_2. Es bietet sich an, die Produktion $A \to \varepsilon$ zu entfernen und gleichzeitig für jede Produktion der Form

$$B \to AC, \quad B \to CA \quad \text{oder} \quad B \to A$$

die neuen Produktionen

$$B \to C, \quad B \to C \quad \text{oder} \quad B \to \varepsilon$$

hinzuzufügen. Aber damit haben wir uns eine neue ε-Produktion eingehandelt.

Schritt 3.1: Bestimme $V' = \{A \in V \mid A \xrightarrow{*} \varepsilon\}$.

Wie? Seien $A_1 \to \varepsilon, \ldots, A_k \to \varepsilon$ sämtliche ε-Produktionen in P_2. Wir setzen anfänglich

$$W = \{A_1, \ldots, A_k\}, \quad V' = \emptyset, \quad P' = P_2$$

(1) Entferne irgendeine Variable A aus der Menge W, und füge A zu V' hinzu. Für jede Produktion der Form

$$B \to AC, \quad B \to CA \quad \text{oder} \quad B \to A \quad (\text{mit } B \neq A)$$

füge die neuen Produktionen $B \to C$, $B \to \varepsilon$ zur Menge P' hinzu. Wenn $B \to \varepsilon$ zu P' hinzugefügt wird, dann füge B zur Menge W hinzu. Um zu verhindern, dass A wieder in die Menge W hinzugefügt wird, entferne alle Produktionen

$$A \to \alpha$$

von P'.

(2) Wiederhole (1), solange $W \neq \emptyset$.

Schritt 3.2: Für jede Regel

$$A \to BC \quad \text{oder} \quad A \to CB \quad \text{mit} \quad B \in V'$$

füge die Regel $A \to C$ zu P_2 hinzu und streiche alle ε-Produktionen. Nenne die neue Produktionenmenge P_3.

Schritt 4: Die Produktionen in P_3 sind von der Form

$$A \to a, \quad A \to B, \quad A \to BC,$$

und nur die *Kettenregeln* $A \to B$ sind noch zu entfernen.

Schritt 4.1: Um die Kettenregeln besser veranschaulichen zu können, betrachten wir den Graphen $H(P_3)$:

- Die Knoten von $H(P_3)$ entsprechen den Variablen,
- für jede Kettenregel $A \to B$ erhält $H(P_3)$ die entsprechende Kante.

Mit einer Tiefensuche auf dem Graphen $H(P_3)$ können wir alle Kreise des Graphen entdecken: Ein Kreis wird nämlich durch Rückwärtskanten geschlossen. Wähle einen beliebigen Kreis $A_1 \to A_2 \to \ldots \to A_k \to A_1$. Jetzt ersetze systematisch in jeder Produktion jede Variable $A_2, \ldots, A_k$ durch A_1; dabei entstehende Produktionen $A_1 \to A_i$ werden entfernt.

Die Variablen $A_2, \ldots, A_k$ sind jetzt überflüssig, da sie in keiner Produktion mehr auftauchen. Wir wiederholen dieses Verfahren, bis alle Kreise zerstört sind. Die neue Produktionenmenge sei P_4.

Schritt 4.2: Der Graph $H(P_4)$ ist jetzt kreisfrei. Wir numerieren die Knoten mit der Methode des topologischen Sortierens, so dass

$$A_i \to A_j$$

bedingt, dass $i < j$. Seien $A_1, \ldots, A_m$ die Variablen.

Es gibt offensichtlich keine Kettenregel der Form $A_m \to A_i$. Wir betrachten nun sämtliche unmittelbaren Vorgänger von A_m. Existiert eine Produktion $A_k \to A_m$, so entfernen wir diese Produktion und fügen für jede Produktion $A_m \to \alpha$ die Produktion $A_k \to \alpha$ hinzu.

α ist aber entweder ein Buchstabe oder von der Form BC. Wir wiederholen dieses Verfahren, bis alle Kettenregeln entfernt sind. Jetzt haben wir Chomsky-Normalform erreicht. □

Bemerkung 3.1 Wir haben mehr erreicht als im Satz ausgesagt. Wir haben gezeigt, wie aus einer kontextfreien Grammatik G eine äquivalente Grammatik in Chomsky-Noormalform *effizient* konstruiert werden kann. Weiterhin ist die Größe der Grammatik, also die Gesamtlänge aller Produktionen, nur polynomiell angestiegen.

Wir spielen nun den kompletten Algorithmus schrittweise an einer Fallstudie durch. Unsere Ausgangsgrammatik sei $G = (\Sigma, V, S, P)$ mit $\Sigma = \{a, b, c\}$ und $V = \{S, A, B, C\}$. Die Produktionsmenge P ist:

$$
\begin{array}{r|lcl}
1 & S & \rightarrow & aABCb \\
2 & S & \rightarrow & \varepsilon \\
3 & S & \rightarrow & C \\
4 & A & \rightarrow & ABC \\
5 & A & \rightarrow & B \\
6 & A & \rightarrow & \varepsilon \\
7 & A & \rightarrow & cC \\
8 & B & \rightarrow & aSc \\
9 & B & \rightarrow & aC \\
10 & B & \rightarrow & S \\
11 & B & \rightarrow & b \\
12 & C & \rightarrow & cC \\
13 & C & \rightarrow & SCCB \\
14 & C & \rightarrow & \varepsilon
\end{array}
$$

Im **ersten Schritt** haben wir die Regeln 1, 7, 8, 9 und 12 zu bearbeiten. Wir führen die neuen Variablen X_a, X_b und X_c ein. Die betreffenden Produktionen lauten nun

$$
\begin{array}{r|lcl}
1' & S & \rightarrow & X_aABCX_b \\
7' & A & \rightarrow & X_cC \\
8' & B & \rightarrow & X_aSX_c \\
9' & B & \rightarrow & X_aC \\
12' & C & \rightarrow & X_cC
\end{array}
$$

und weisen rechts nur noch Strings aus V^* auf. Dazu kommen die neuen Produktionen

$$
\begin{array}{r|lcl}
15 & X_a & \rightarrow & a \\
16 & X_b & \rightarrow & b \\
17 & X_c & \rightarrow & c
\end{array} .
$$

Im **zweiten Schritt** widmen wir uns den Produktionen, die auf Variablenstrings abbilden, die länger als 2 sind. Wir teilen die Produktionen 1', 4, 8', 13 auf. Danach lautet unsere Produktionsmenge P_2:

$1'a$	S	$\rightarrow$	$X_aX_1^1$
$1'b$	X_1^1	$\rightarrow$	AX_2^1
$1'c$	X_2^1	$\rightarrow$	BX_3^1
$1'd$	X_3^1	$\rightarrow$	CX_b
2	S	$\rightarrow$	ε
3	S	$\rightarrow$	C
$4a$	A	$\rightarrow$	AX_1^4
$4b$	X_1^4	$\rightarrow$	BC
5	A	$\rightarrow$	B
6	A	$\rightarrow$	ε
$7'$	A	$\rightarrow$	X_cC
$8'a$	B	$\rightarrow$	$X_aX_1^8$
$8'b$	X_1^8	$\rightarrow$	SX_c
$9'$	B	$\rightarrow$	X_aC
10	B	$\rightarrow$	S
11	B	$\rightarrow$	b
$12'$	C	$\rightarrow$	X_cC
$13a$	C	$\rightarrow$	SX_1^{13}
$13b$	X_1^{13}	$\rightarrow$	CX_2^{13}
$13c$	X_2^{13}	$\rightarrow$	CB
14	C	$\rightarrow$	ε
15	X_a	$\rightarrow$	a
16	X_b	$\rightarrow$	b
17	X_c	$\rightarrow$	c

Jetzt haben wir uns den ε-Produktionen in **Schritt drei** zu widmen. Wir müssen die Menge aller Variablen bestimmen, die auf ε abgebildet werden können. (Schritt 3.1). Wir stellen den Ablauf tabellarisch dar.

Ausgangssituation: $W = \{S, A, C\}, V' = \emptyset, P' = P_2$

Phase 1, bewege S aus W nach V'	
hinzuzufügen	$X_1^8 \rightarrow X_c$
	$B \rightarrow \varepsilon$
	$C \rightarrow X_1^{13}$
füge B zu W hinzu.	
entfernen	$S \rightarrow X_aX_1^1$
	$S \rightarrow \varepsilon$
	$S \rightarrow C$
$W = \{A, C, B\}$ $V' = \{S\}$	

Phase 2, bewege A aus W nach V'	
hinzuzufügen	$X_1^1 \rightarrow X_2^1$
	$A \rightarrow X_1^4$
W bleibt unverändert	
entfernen	$A \rightarrow AX_1^4$
	$A \rightarrow B$
	$A \rightarrow \varepsilon$
	$A \rightarrow X_cC$
	$A \rightarrow X_1^4$
$W = \{C, B\}$ $V' = \{S, A\}$	

Phase 3, bewege C aus W nach V'	
hinzuzufügen	$X_3^1 \rightarrow X_b$ $X_1^4 \rightarrow B$ $B \rightarrow X_a$ $C \rightarrow X_c$ $X_1^{13} \rightarrow X_2^{13}$ $X_2^{13} \rightarrow B$
W bleibt unverändert	
entfernen	$C \rightarrow cC$ $C \rightarrow SX_1^{13}$ $C \rightarrow \varepsilon$ $C \rightarrow X_1^{13}$ $C \rightarrow X_c$
$W = \{B\}$	$V' = \{S, A, C\}$

Phase 4, bewege B aus W nach V'	
hinzuzufügen	$X_2^1 \rightarrow X_3^1$ $X_1^4 \rightarrow C$ $X_2^{13} \rightarrow C$ $X_1^4 \rightarrow \varepsilon$ $X_2^{13} \rightarrow \varepsilon$
$W := W \cup \{X_1^4, X_2^{13}\}$	
entfernen	$B \rightarrow X_a X_1^8$ $B \rightarrow X_a C$ $B \rightarrow S$ $B \rightarrow b$ $B \rightarrow \varepsilon$ $B \rightarrow X_a$
$W = \{X_1^4, X_2^{13}\}$	$V' = \{S, A, C, B\}$

Phase 5, bewege X_1^4 aus W nach V'	
hinzuzufügen	*nichts*
W bleibt unverändert	
entfernen	$X_1^4 \rightarrow BC$ $X_1^4 \rightarrow B$ $X_1^4 \rightarrow C$ $X_1^4 \rightarrow \varepsilon$
$W = \{X_2^{13}\}$	$V' = \{S, A, C, B, X_1^4\}$

Phase 6, bewege X_2^{13} aus W nach V'	
hinzuzufügen	$X_1^{13} \rightarrow C$ $X_1^{13} \rightarrow \varepsilon$
$W := W \cup \{X_1^{13}\}$	
entfernen	$X_2^{13} \rightarrow BC$ $X_2^{13} \rightarrow B$ $X_2^{13} \rightarrow C$ $X_2^{13} \rightarrow \varepsilon$
$W = \{X_1^{13}\}$	$V' = \{S, A, C, B, X_1^4, X_2^{13}\}$

Phase 7, bewege X_1^{13} aus W nach V'	
hinzuzufügen	*nichts*
$W = \emptyset$	
entfernen	$X_1^{13} \rightarrow CX_2^{13}$ $X_1^{13} \rightarrow X_2^{13}$ $X_1^{13} \rightarrow C$ $X_1^{13} \rightarrow \varepsilon$
$W = \emptyset$	$V' = \{S, A, C, B, X_1^4, X_2^{13}, X_1^{13}\}$

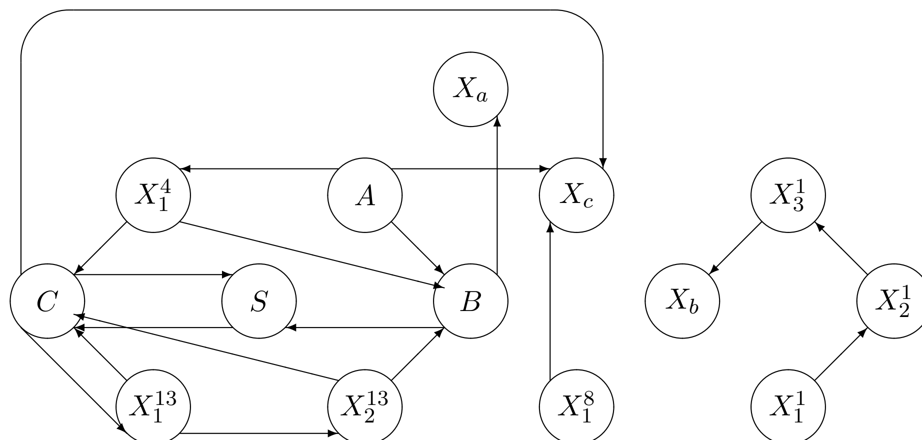
Als Menge V' erhalten wir also $\{S, A, B, C, X_1^4, X_1^{13}, X_2^{13}\}$. Das ist die Menge all jener Variablen, die auf ε abgebildet werden können.

Gemäß **Schritt 3.2** können wir nun die Produktionenmenge P_3 bestimmen. Wir stellen P_2 und P_3 in der folgenden Tabelle gegenüber.

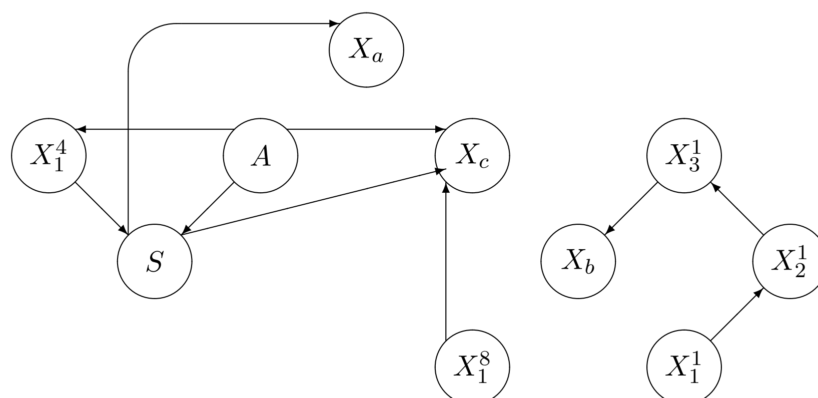
	P_2	P_3
$1'a$	$S{\to}X_aX_1^1$	$S{\to}X_aX_1^1$
$1'b$	$X_1^1{\to}AX_2^1$	$X_1^1{\to}AX_2^1$ $X_1^1{\to}X_2^1$
$1'c$	$X_2^1{\to}BX_3^1$	$X_2^1{\to}BX_3^1$ $X_2^1{\to}X_3^1$
$1'd$	$X_3^1{\to}CX_b$	$X_3^1{\to}CX_b$ $X_3^1{\to}X_b$
2	$S{\to}\varepsilon$	
3	$S{\to}C$	$S{\to}C$
$4a$	$A{\to}AX_1^4$	$A{\to}AX_1^4$ $A{\to}X_1^4$
$4b$	$X_1^4{\to}BC$	$X_1^4{\to}BC$ $X_1^4{\to}B$ $X_1^4{\to}C$
5	$A{\to}B$	$A{\to}B$
6	$A{\to}\varepsilon$	
$7'$	$A{\to}X_cC$	$A{\to}X_cC$ $A{\to}X_c$
$8'a$ $8'b$	$B{\to}X_aX_1^8$ $X_1^8{\to}SX_c$	$B{\to}X_aX_1^8$ $X_1^8{\to}SX_c$ $X_1^8{\to}X_c$
$9'$	$B{\to}X_aC$	$B{\to}X_aC$ $B{\to}X_a$
10	$B{\to}S$	$B{\to}S$
11	$B{\to}b$	$B{\to}b$
$12'$	$C{\to}X_cC$	$C{\to}cC$ $C{\to}X_c$
$13a$	$C{\to}SX_1^{13}$	$C{\to}SX_1^{13}$ $C{\to}S$ $C{\to}X_1^{13}$
$13b$	$X_1^{13}{\to}CX_2^{13}$	$X_1^{13}{\to}CX_2^{13}$ $X_1^{13}{\to}C$ $X_1^{13}{\to}X_2^{13}$
$13c$	$X_2^{13}{\to}CB$	$X_2^{13}{\to}CB$ $X_2^{13}{\to}C$ $X_2^{13}{\to}B$
14	$C{\to}\varepsilon$	
15	$X_a{\to}a$	$X_a{\to}a$
16	$X_b{\to}b$	$X_b{\to}b$
17	$X_c{\to}c$	$X_c{\to}c$

Beachte, dass wir hier entstehende neue ε-Regeln, wie sie formal etwa in Zeile $4b$ entstehen würden ($X_1^4 \to \varepsilon$) gleich unterdrückt haben. Auch die in Zeile $4a$ entstehende triviale Regel $A \to A$ haben wir gar nicht erst aufgenommen.

Für den abschließenden **vierten Schritt** benötigen wir nunmehr den Graphen, der die Kettenregeln der Grammatik darstellt. Man vergleiche:



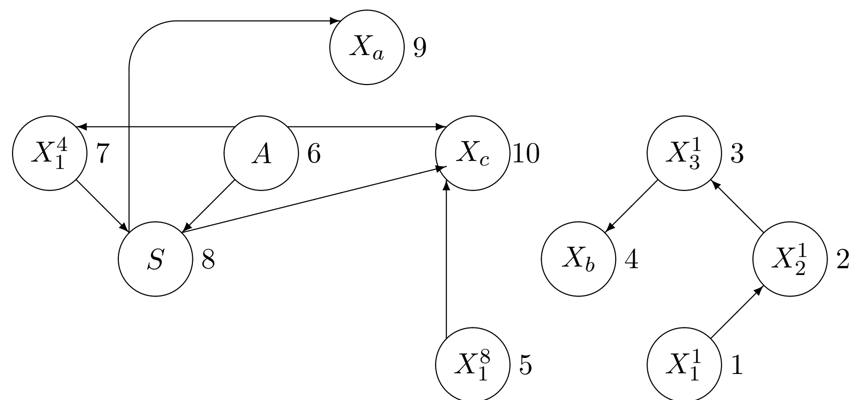
Mittels eines gängigen Graphenalgorithmus wie etwa der Tiefensuche ermittelt man die Kreise. Nehmen wir an, wir würden Kreis $B, S, C, X_1^{13}, X_2^{13}$ finden, dann können wir diese fünf Variablen nun zu einer zusammenfassen. Da das Startsymbol unter ihnen ist, wählen wir dieses. Wir erhalten:



Der Kreis ist nun zykelfrei. Unsere Produktionenmenge P_3 verkleinert sich zu P_4:

P_4
$S \to X_a X_1^1$
$X_1^1 \to A X_2^1$
$X_1^1 \to X_2^1$
$X_2^1 \to S X_3^1$
$X_2^1 \to X_3^1$
$X_3^1 \to S X_b$
$X_3^1 \to X_b$
$A \to A X_1^4$
$A \to X_1^4$
$X_1^4 \to SS$
$X_1^4 \to S$
$A \to S$
$A \to X_c S$
$A \to X_c$
$S \to X_a X_1^8$
$X_1^8 \to S X_c$
$X_1^8 \to X_c$
$S \to X_a S$
$S \to a$
$S \to b$
$S \to X_c C$
$S \to c$
$S \to SS$
$S \to X_a$
$s \to X_c$
$X_a \to a$
$X_b \to b$
$X_c \to c$

Eine topologische Sortierung kann z.B. so aussehen:



Wir beseitigen nun sukzessive die Kettenregeln in der durch die Sortierung vorgegebenen Reihenfolge. Unsere Grammatik wächst noch einmal. Aber das Ziel ist damit auch erreicht.

statt	*neu*
$S \to X_c$	$S \to c$ ex. bereits
$X_1^8 \to X_c$	$X_1^8 \to c$
$A \to X_c$	$A \to c$
$S \to X_a$	$S \to a$ ex. bereits
$X_1^4 \to S$	$X_1^4 \to X_a X_1^1$
	$X_1^4 \to X_a X_1^8$
	$X_1^4 \to X_a S$
	$X_1^4 \to a$
	$X_1^4 \to b$
	$X_1^4 \to X_c C$
	$X_1^4 \to c$
	$X_1^4 \to SS$ ex. bereits
$A \to S$	$A \to X_a X_1^1$
	$A \to X_a X_1^8$
	$A \to X_a S$
	$A \to a$
	$A \to b$
	$A \to X_c C$
	$A \to c$ ex. bereits
	$A \to SS$
$A \to X_1^4$	$A \to SS$ ex. bereits
	$A \to X_a X_1^1$ ex. bereits
	$A \to X_a X_1^8$ ex. bereits
	$A \to X_a S$ ex. bereits
	$A \to a$ ex. bereits
	$A \to b$ ex. bereits
	$A \to X_c C$ ex. bereits
	$A \to c$ ex. bereits
$X_3^1 \to X_b$	$X_3^1 \to b$
$X_2^1 \to X_3^1$	$X_2^1 \to S X_b$
	$X_2^1 \to b$
$X_1^1 \to X_2^1$	$X_1^1 \to S X_3^1$
	$X_1^1 \to S X_b$
	$X_1^1 \to b$

Diese Grammatik ist nun in Chomsky-Normalform.

Wir können jetzt das Compilerproblem (bzw. das Wortproblem) effizient lösen.

Satz 3.6 **Satz von Cocke, Younger und Kasami**

Sei G eine Grammatik in Chomsky-Normalform. Dann kann in Zeit

$$O\left(|\textit{Produktionen von } G| \cdot |w|^3\right)$$

entschieden werden, ob $w \in L(G)$.

Beweis: Wir benutzen die Methode der dynamischen Programmierung. Die Grammatik $G = (\Sigma, V, S, P)$ liege in in Chomsky-Normalform vor und $w = w_1 \cdots w_n \in \Sigma^n$ sei die Eingabe.

Für jedes i, j $(1 \leq i \leq j \leq n)$ möchten wir die Menge

$$V_{i,j} = \left\{ A \in V \mid A \stackrel{*}{\to} w_i \cdots w_j \right\}$$

bestimmen. (Beachte, dass $w \in L(G)$ genau dann gilt, wenn $S \in V_{1,n}$.)

(1) Die Mengen $V_{i,i}$ können sofort bestimmt werden, da

$$V_{i,i} = \left\{ A \in V \mid A \stackrel{*}{\to} w_i \right\} = \left\{ A \in V \mid A \to w_i \right\},$$

denn G ist in Chomsky-Normalform.

(2) Angenommen, alle Mengen $V_{i,j}$ mit $j - i < s$ sind bestimmt. Seien jetzt i und j vorgegeben mit $j - i = s$. Es gilt

$$\begin{aligned} A \in V_{i,j} &\Leftrightarrow A \stackrel{*}{\to} w_i \cdots w_j \\ &\Leftrightarrow \text{Es gibt eine Produktion } A \to BC \text{ mit} \\ & B \stackrel{*}{\to} w_i \cdots w_k, \text{ und } C \stackrel{*}{\to} w_{k+1} \cdots w_j \\ &\Leftrightarrow \text{Es gibt eine Produktion } A \to BC \text{ mit} \\ & B \in V_{i,k} \text{ und } C \in V_{k+1,j}. \end{aligned}$$

Wir können jetzt $V_{i,j}$ in Zeit höchstens

$$n \cdot \text{Anzahl der Produktionen}$$

bestimmen. Die behauptete Laufzeit folgt, da $\binom{n}{2}$ Mengen zu bestimmen sind. □

Kubische Laufzeit ist für Anwendungen nicht tolerabel. Wir werden uns deshalb später auf determinstisch kontextfreie Sprachen beschränken.

3.3 Das Pumping Lemma und Ogden's Lemma

Kontextfreie Grammatiken scheinen recht mächtig zu sein und die Frage nach den Grenzen ihrer Beschreibungskraft drängt sich auf: Wie sehen möglichst einfache, nicht-kontextfreie Sprachen aus? Für die Beantwortung dieser Frage leiten wir zuerst ein Pumping-Lemma für kontextfreie Sprachen her:

Satz 3.7 *Sei L eine kontextfreie Sprache. Dann gibt es eine Pumpingkonstante N, so dass jedes $z \in L$ mit $|z| \geq N$ eine Zerlegung $z = uvwxy$ besitzt mit den Eigenschaften*

(a) $|vwx| \leq N$ und $|vx| \geq 1$ und

(b) $uv^i wx^i y \in L$ für alle $i \geq 0$.

Wir betrachten zuerst zwei Beispiele:

Beispiel 3.5 Wir wollen zeigen, dass

$$L = \left\{ a^m b^m c^m \mid m \geq 0 \right\}$$

nicht kontextfrei ist. Den Beweis führen wir durch eine Anwendung des Pumping-Lemmas.

Sei N die (unbekannte) Pumpingkonstante. Wir wählen dann das Wort $z = a^N b^N c^N \in L$. Wenn L kontextfrei ist, dann gibt es eine Zerlegung

$$z = uvwxy$$

(mit $|vwx| \leq N$ und $|vx| \geq 1$, so dass $uv^i wx^i y \in L$ für jedes $i \geq 0$. Da $|vwx| \leq N$, enthält vwx nur a's und b's oder nur b's und c's. Wir nehmen das Erstere an. (Beachte, dass es auch möglich ist, dass vwx nur a's, nur b's oder nur c's enthält. Diese Fälle werden aber durch die Erstgenannten mit abgedeckt.)

Dann ist aber $uv^2wx^2y \notin L$, denn das aufgepumpte Wort enthält mindestens ein a oder mindestens ein b mehr als es c's enthält. Also ist L nicht kontextfrei.

Beispiel 3.6 Wir möchten zeigen, dass die Sprache

$$L = \left\{ d^r a^x b^y c^z \mid r = 0 \text{ oder } x = y = z \right\}$$

nicht kontextfrei ist. Wir versuchen, das Pumping-Lemma anzuwenden. Für die Pumpingkonstante N sei $z \in L$ ein Wort mit $|z| \geq N$.

Fall 1: d kommt nicht in z vor.

Dann wird d auch nach dem Auf- oder Abpumpen nicht im Wort vorkommen, und das auf- oder abgepumpte Wort gehört deshalb zur Sprache L.

Fall 2: d kommt in z vor.

Dann werden wir allerdings nicht mit der Zerlegung

$$u = v = w = \varepsilon, \quad x = d \quad \text{und} \quad y = d^{r-1} a^x b^x c^x$$

„fertig": Auf- oder Abpumpen wird nur die Anzahl von d's variieren.

Eine Argumentation mit dem Pumping-Lemma mißlingt also. Wir benötigen stattdessen eine Version des Pumping-Lemmas, die es uns erlaubt, mindestens ein a, b oder c aufpumpen zu können. Dies wird erreicht durch

Satz 3.8 **Ogden's Lemma**

Sei L eine kontextfreie Sprache. Dann gibt es eine Pumpingkonstante N, so dass jedes $z \in L$ mit

mindestens N markierten Buchstaben

eine Zerlegung $z = uvwxy$ besitzt, wobei

(a) höchstens N Buchstaben in vwx markiert sind,

(b) mindestens ein Buchstabe in vx markiert ist und

(c) $uv^i wx^i y \in L$ für alle $i \geq 0$ gilt.

Bemerkung 3.2 **(a)** Beachte, dass das Pumping-Lemma aus Odgen's Lemma folgt, wenn wir alle Buchstaben von z markieren.

(b) Wir können jetzt zeigen, dass

$$\left\{ d^r a^x b^y c^z \mid r = 0 \text{ oder } x = y = z \right\}$$

nicht kontextfrei ist. Wenn N die Pumpingkonstante ist, wähle

$$z = da^N b^N c^N$$

und markiere $a^N b^N c^N$. Dann besitzt vwx kein a oder kein c (denn vwx besitzt nur n markierte Buchstaben). Nach dem Aufpumpen wird uv^2wx^2y also zuwenige a's oder zuwenige c's besitzen.

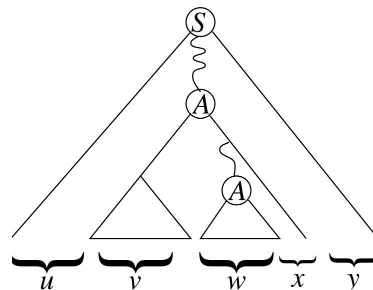
Beweis von Satz 3.8: Wenn L kontextfrei ist, dann ist auch $L \setminus \{\varepsilon\}$ kontextfrei, und es genügt, Odgen's Lemma für $L \setminus \{\varepsilon\}$ zu beweisen. Deshalb können wir annehmen, dass

$$L \setminus \{\varepsilon\} = L(G)$$

für eine Grammatik $G = (\Sigma, V, S, P)$ in Chomsky-Normalform ist. Wir wählen

$$N = 2^{|V|+1}$$

als Pumpingkonstante. Sei nun $z \in L$ ein beliebiges Wort in L mit mindestens N markierten Buchstaben. Wir betrachten einen Ableitungsbaum B für z:



Wir sagen, dass ein Knoten v von B ein Verzweigungsknoten ist, wenn v markierte Blätter sowohl im linken wie auch im rechten Teilbaum besitzt. Wir erhalten die folgenden Eigenschaften von B:

(a) B ist binär, da G in Chomsky-Normalform vorliegt.

(b) B hat $2^{|V|+1}$ markierte Blätter, also Blätter, die einen markierten Buchstaben speichern.

(c) Es gibt einen Weg W von der Wurzel zu einem Blatt, so dass W mindestens $|V| + 1$ Verzweigungsknoten besitzt.

Die Eigenschaften (a) und (b) sind offensichtlich. Wir weisen Eigenschaft (c) nach. Konstruiere einen Weg W so, dass W jedes Mal mit dem Kind mit den meisten markierten Blättern fortgesetzt wird. W beginnt an der Wurzel mit mindestens N markierten Blättern. Da die Anzahl der markierten Blätter sich nur bei einem Verzweigungsknoten verringert, und dann auch nur um höchstens den Faktor 2, muß W mindestens $\log_2 N = |V| + 1$ Verzweigungsknoten besitzen.

Unter den Verzweigungsknoten von W betrachten wir die letzten $|V| + 1$ Verzweigungsknoten, unter denen dann auch eine Variable A zweimal vorkommt. Das erste Vorkommen von A benutzen wir zur Definition von v (Konkatenation der Blätter im linken Teilbaum) und wx (Konkatenation der Blätter im rechten Teilbaum).

Beachte, dass v (und damit auch vx) mindestens einen markierten Buchstaben besitzt. Weiterhin besitzt vwx höchstens $2^{|V|+1} = N$ markierte Buchstaben: Wenn vwx mehr als N markierte

Buchstaben besitzt, dann besitzt W (auf seinem Endstück) auch mehr als $|V|+1$ Verzweigungsknoten. Damit gilt

$$S \xrightarrow{*} uAy$$

sowie

$$A \xrightarrow{*} vAx \quad \text{und} \quad A \xrightarrow{*} w.$$

Deshalb sind auch die Ableitungen

$$S \xrightarrow{*} uAy \xrightarrow{*} uwy = uv^0wx^0y$$

wie auch

$$S \xrightarrow{*} uAy \xrightarrow{*} uvAxy \xrightarrow{*} uv^2Ax^2y$$

$$\xrightarrow{*} \ldots \xrightarrow{*} uv^iAx^iy \xrightarrow{*} uv^iwx^iy$$

möglich, und deshalb ist $uv^iwx^iy \in L$ für alle $i \geq 0$. □

Aufgabe 52
Eine Grammatik heißt linear, wenn jede Produktion in P die Form $u \Rightarrow xvy$, für $u, v \in V$ und $x, y \in \Sigma^*$ oder $u \Rightarrow \varepsilon$ hat.

Entwirf einen Algorithmus, der das Wortproblem für eine fest vorgegebene lineare Grammatik $G = (\Sigma, V, S, P)$ löst. Der Algorithmus sollte in Zeit $O(|w|^2)$ entscheiden, ob $w \in L(G)$ ist.

Hinweis: Man kann die dynamische Programmierung benutzen.

Aufgabe 53
Gegeben sei eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$ in Chomsky-Normalform. Außerdem sei eine Kostenfunktion über den Produktionen P der Grammatik gegeben, die jeder Produktion $p \in P$ einen Kostenfaktor $c(p) \in \mathbb{N}$ zuordnet.
Die *Kosten einer Ableitung* $S \overset{*}{\Rightarrow} w$ sind die Summe der Kosten der benutzen Produktionen.

Beschreibe einen *möglichst* effizienten Algorithmus, der für ein Wort $w \in \Sigma^*$ den Wert einer Ableitung mit minimalen Kosten bestimmt (bzw. den Wert unendlich ausgibt, falls $w \notin L(G)$). **Bestimme** die Laufzeit deines Algorithmus.

Aufgabe 54
Welche der folgenden Sprachen über dem Alphabet $\Sigma = \{a, b, c\}$ sind kontextfrei?
Entwirf eine kontextfreie Grammatik, falls eine Sprache kontextfrei ist; **benutze** das Pumping-Lemma für kontextfreie Sprachen oder Ogden's Lemma, um zu zeigen, dass eine Sprache nicht kontextfrei ist.

$L_1 = \{a^ib^ja^jb^i \mid i, j \geq 0\}$,

$L_2 = \{a^ib^ja^ib^j \mid i, j \geq 0\}$,

$L_3 = \{a^ib^ic^j \mid i \neq j\}$,

$L_4 = \{a^ib^jc^k \mid i \leq k \text{ oder } j \leq k\}$,

$L_5 = \{a^ib^jc^k | i = j = k \text{ gilt nicht}\}$,

$L_6 = \{a^ib^jc^{i \cdot j} | i, j \geq 0\}$,

$L_7 = \{a^ib^ic^jd^j | i, j \geq 0\}$.

Aufgabe 55
Wir betrachten in dieser Aufgabe eine vereinfachte Klasse von „Programmiersprachen“. Programme der Sprache müssen durch die folgende kontextfreie Grammatik G erzeugbar sein.
Es sei $G = \{\Sigma, V, S, P\}$, mit $\Sigma = \{REAL, INT, x, y, 0, 1, +, -, ;, ., e\}$,

$V = \{S, D, A, < decl >, < type >, < var\text{-}list >, < id >, < integer >, < sign >, < digit >, < real >\}$ und den Produktionen:

$$\begin{array}{lll} S & \Rightarrow & DA \\ D & \Rightarrow & \varepsilon \mid D < decl > \\ A & \Rightarrow & \varepsilon \mid A < id >=< integer >; \mid A < id >=< real >; \\ < decl > & \Rightarrow & < type >< var\text{-}list >; \\ < type > & \Rightarrow & REAL \mid INT \\ < var\text{-}list > & \Rightarrow & < id > \mid < var\text{-}list >, < id > \\ < id > & \Rightarrow & x \mid y \mid < id > x \mid < id > y \\ < integer > & \Rightarrow & < sign >< digit > \\ < sign > & \Rightarrow & \varepsilon \mid + \mid - \\ < digit > & \Rightarrow & < digit > 0 \mid < digit > 1 \mid 0 \mid 1 \\ < real > & \Rightarrow & < sign >< digit > . < digit > e < sign >< digit > \end{array}$$

Zusätzlich soll für unsere Programmiersprache folgendes gelten: Jede im Anweisungsteil A benutzte Variable muß zuvor im Deklarationsteil D deklariert sein, und der Typ der im Anweisungsteil benutzten Variablen muß mit dem im Deklarationsteil festgelegten Typen übereinstimmen.

Zeige, dass diese Programmiersprache nicht kontextfrei ist.

Anmerkung: Wie wir an dieser Aufgabe sehen, ist folgende Arbeitsweise von Compilern notwendig: Zunächst wird der Deklarationsteil abgearbeitet, indem die Variablen in eine Symboltabelle gehasht werden. Danach können auf dem verbliebenen Programm (ohne Deklarationsteil) die effizienten Lösungen des Compilerproblems für kontextfreie Sprachen angewendet werden.

Aufgabe 56
Eine Grammatik heißt linear, wenn jede Produktion in P die Form $u \Rightarrow xvy$, für $u, v \in V$ und $x, y \in \Sigma^*$ oder $u \Rightarrow \varepsilon$ hat. Wir nennen eine Sprache L linear, wenn es eine lineare Grammatik G mit $L = L(G)$ gibt.

(a) **Beweise** das folgende Pumping-Lemma für lineare Sprachen: Wenn L eine lineare Sprache ist, dann gibt es eine Pumping-Konstante N, so dass sich jedes $z \in L$ mit $|z| \geq N$ als $z = uvwxy$ schreiben lässt, wobei $|uvxy| \leq N, |vx| \geq 1$ und $uv^iwx^iy \in L$ für alle $i \geq 0$ gilt.

(b) **Zeige**, dass $\{a^ib^ic^jd^j | i, j \geq 0\}$ keine lineare Sprache ist.

Aufgabe 57
Sei G eine Grammatik in Chomsky-Normalform.

(a) Sei $L(G)$ endlich. **Gib** eine obere Schranke für die Länge eines längsten Wortes in $L(G)$ an.

(b) Sei $L(G)$ unendlich. **Gib** eine obere Schranke für die Länge eines kürzesten Wortes in $L(G)$ an.

3.3.1 Abschlusseigenschaften kontextfreier Sprachen

Als Konsequenz von Beispiel 3.5 erhalten wir, dass kontextfreie Sprachen nicht unter Durchschnittsbildung und auch nicht unter Komplementbildung abgeschlossen sind:

Satz 3.9 *(a) Wenn L_1 und L_2 kontextfrei sind, dann sind auch $L_1 \cup L_2$, $L_1 \circ L_2$ und L_1^* kontextfrei.*

(b) Es gibt kontextfreie Sprachen L_1 und L_2, so dass der Durchschnitt $L_1 \cap L_2$ nicht kontextfrei ist.

(c) Es gibt eine kontextfreie Sprache L, so dass die Komplementsprache $\overline{L}$ nicht kontextfrei ist.

Beweis: (a) Es gelte $L_1 = L(G_1)$ und $L_2 = L(G_2)$, wobei $G_1 = (\Sigma, V_1, S_1, P_1)$, $G_2 = (\Sigma, V_2, S_2, P_2)$ und $V_1 \cap V_2 = \emptyset$. Setze

- $V = V_1 \cup V_2 \cup \{S\}$ mit $S \notin V_1 \cup V_2$
- $P = P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}$

für ein neues Symbol S. Dann ist offensichtlich $L_1 \cup L_2 = L(G)$, wobei $G = (\Sigma, V, S, P)$. Wenn wir

$$P = P_1 \cup P_2 \cup \Big\{S \to S_1 S_2\Big\}$$

setzen, erhalten wir $L(G) = L_1 \circ L_2$.

Schließlich erzeugt die Grammatik ${G_1}^* = (\Sigma, V_1, S_1, {P_1}^*)$ mit ${P_1}^* = P_1 \cup \{S_1 \to S_1 S_1,\ S_1 \to \varepsilon\}$ den Kleene-Abschluß ${L_1}^*$.

(b) Wir setzen

$$L_1 = \Big\{a^n b^n c^m \,|\, n, m \geq 0\Big\}$$

und

$$L_2 = \Big\{a^m b^n c^n \,|\, n, m \geq 0\Big\}.$$

Beide Sprachen sind kontextfrei, denn zum Beispiel besitzt L_1 die Grammatik

$$\begin{array}{rcl} S & \to & Sc \,|\, T \\ T & \to & aTb \,|\, \varepsilon. \end{array}$$

Aber $L_1 \cap L_2$ ist nicht kontextfrei, denn es ist

$$L_1 \cap L_2 = \Big\{a^n b^n c^n \,|\, n \geq 0\Big\}.$$

(c) Angenommen, mit L ist auch stets die Komplementsprache $\overline{L}$ kontextfrei. Wegen

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

wären die kontextfreien Sprachen dann aber unter der Durchschnittsbildung abgeschlossen. □

Nur wenige Entscheidungsprobleme lassen sich effizient für kontextfreie Sprachen lösen. Das Leerheitsproblem „Ist $L(G)$ leer“ ist eine rühmliche Ausnahme:

Aufgabe 58
Gegeben sei eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$ in Chomsky-Normalform. **Beschreibe** einen *möglichst* effizienten Algorithmus, der entscheidet, ob die von G erzeugte Sprache $L(G)$ leer ist. **Bestimme** die Laufzeit deines Algorithmus.

3.4 Kellerautomaten

Wir möchten wie im Fall regulärer Sprachen ein Maschinenmodell entwerfen, das genau die Klasse kontextfreier Sprachen akzeptiert. Neben einer äquivalenten Charakterisierung kontextfreier Sprachen bieten Kellerautomaten die Möglichkeit, *deterministisch kontextfreie Sprachen* einzuführen (nämlich diejenigen Sprachen, die von einem deterministischen Kellerautomaten akzeptiert werden). Die Klasse der deterministisch kontextfreien Sprachen ist immer noch ausdruckskräftig. Aber, und dies ist der große Vorteil, sehr schnelle Algorithmen lösen das Compilerproblem.

Definition 3.10 *Eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$ ist in Greibach-Normalform, wenn alle Produktionen von der Form*

$$A \to a\alpha \quad (\text{für } A \in V,\ a \in \Sigma \text{ und } \alpha \in V^*)$$

sind.

Satz 3.11 *Die Grammatik $G = (\Sigma, V, S, P)$ sei in Chomsky-Normalform. Dann gibt es eine Grammatik $G^* = (\Sigma, V^*, S^*, P^*)$ in Greibach-Normalform, so dass*

- $L(G) = L(G^*)$
- $|P^*| = O\left(|P|^3\right)$.

Beweis: Siehe I. Wegener, Theoretische Informatik, S. 172–181. □

Wir entwerfen jetzt das Maschinenmodell eines Kellerautomaten, das Linksableitungen für eine Grammatik in Greibach-Normalform simulieren kann. Beachte, dass eine Linksableitung eines Wortes $w \in \Sigma^*$ die Form

$$S \to w_1 A_1 \alpha_1 \to w_1 w_2 A_2 \alpha_2 \to w_1 w_2 w_3 A_3 \alpha_3 \stackrel{*}{\to} w_1 \cdots w_n$$

hat. Es liegt deshalb nahe, das Maschinenmodell mit einem sequentiell lesbaren Eingabeband und einem Stack auszustatten: Anfänglich befindet sich nur das Startsymbol S auf dem Stack. Dann wird S gelesen und durch den String $A_1\alpha_1$ ersetzt, während der Lesekopf nachprüft, ob der Buchstabe w_1 gelesen wurde. Ist dies der Fall, wird im nächsten Schritt

- das zuoberst gelesene Symbol A_1 gelesen und durch den String $A_2\alpha_2$ ersetzt, und
- der Lesekopf überprüft, ob w_2 auf dem Eingabeband gelesen wurde.

Diese Schritte werden wiederholt, bis

- alle Buchstaben gelesen wurden und
- der Stack geleert wurde.

Wir werden allerdings auch den Akzeptanzmodus „Akzeptiere durch Zustand“ betrachten.

Definition 3.12 Kellerautomaten

(a) *Ein nichtdeterministischer Kellerautomat (PDA) A besitzt die folgenden Komponenten:*

- *die endliche Zustandsmenge Q,*
- *das Eingabealphabet Σ,*
- *das Stackalphabet Γ,*
- *den Anfangszustand q_0 und*
- *den anfänglichen Stackinhalt $Z_0 \in \Gamma$ sowie*

- *die Zustandsüberführungsfunktion*

$$\delta\colon Q \times \Big(\Sigma \cup \{\varepsilon\}\Big) \times \Gamma \to \mathcal{P}\left(Q \times \Gamma^*\right).$$

Für den Akzeptanzmodus durch Zustände ist zusätzlich noch eine Teilmenge $F \subseteq Q$ von akzeptierenden Zuständen auszuzeichnen.

(b) *Eine Konfiguration von A (für Eingabe $w \in \Sigma^*$) ist ein Tripel*

$$(q, w', \alpha) \quad \textit{mit } q \in Q,\ w' \in \Sigma^* \textit{ und } \alpha \in \Gamma^*.$$

q ist der gegenwärtige Zustand, w' ist der Suffix der noch nicht gelesenen Buchstaben) von w, und α ist ein Stackinhalt. Wie arbeitet A? Wenn

$$(q, w'_1 \cdots w'_m, \alpha_1 \cdots \alpha_s)$$

die gegenwärtige Konfiguration ist und wenn

$$(q', Z_1 \cdots Z_s) \in \delta(q, w'_1, \alpha_1),$$

dann ist

$$(q', w'_2 \cdots w'_m, Z_1 \cdots Z_s \alpha_2 \cdots \alpha_s)$$

eine mögliche Nachfolgekonfiguration. Wenn

$$(q', Z_1 \cdots Z_s) \in \delta(q, \varepsilon, \alpha_1),$$

dann ist

$$(q', w'_1 \cdots w'_m, Z_1 \cdots Z_s \alpha_2 \cdots \alpha_s)$$

eine mögliche Nachfolgekonfiguration. Alle möglichen Nachfolgekonfigurationen werden auf diese Art und Weise erhalten.

(c) *Akzeptanz durch leeren Stack:*

A akzeptiert die Eingabe $w = w_1 \cdots w_n$ genau dann, wenn es eine Konfigurationenfolge

$$(q_0, w_1 \cdots w_n, Z_0), \ldots, (q, \varepsilon, \varepsilon)$$

gibt. Bis auf den letzten Schritt darf der Keller zu keinem anderen Zeitpunkt leer sein.

(d) *Akzeptanz durch akzeptierende Zustände:*

A akzeptiert die Eingabe $w = w_1 \cdots w_n$ genau dann, wenn es eine Konfigurationenfolge

$$(q_0, w_1 \cdots w_n, Z_0), \ldots, (q, \varepsilon, \alpha)$$

gibt mit $\alpha \in \Gamma^*$ und $q \in F$. Bis auf den letzten Schritt darf auch diesmal der Keller nicht leer werden.

Beispiel 3.7 $L = \{ww^{\text{reverse}} \mid w \in \{a, b\}^*\}$.

Wir entwerfen einen Kellerautomaten für L. Die Idee ist, dass sich der Kellerautomat in einer ersten Phase die gelesenen Zeichen auf dem Stack merkt und irgendwann nichtdeterministisch in eine zweiten Phase übergeht, in der die verbleibenden Zeichen der Eingabe mit dem Inhalt des Stacks verglichen werden. Der Kellerautomat *rät* also die Mitte des Wortes.

Wir wählen $Q = \{\texttt{push}, \texttt{pop}, \texttt{fail}\}$, $\Gamma = \{a, b, Z_0\}$, $q_0 = \texttt{push}$ und die Befehle

$$\begin{array}{rcl}
\delta(\texttt{push}, a, x) & = & \{(\texttt{push}, ax), (\texttt{pop}, ax)\} \\
\delta(\texttt{push}, b, x) & = & \{(\texttt{push}, bx), (\texttt{pop}, bx)\} \\
\delta(\texttt{pop}, a, x) & = & \begin{cases} \{(\texttt{pop}, \varepsilon)\} & a = x \\ \{(\texttt{fail}, x)\} & a \neq x \end{cases} \\
\delta(\texttt{pop}, b, x) & = & \begin{cases} \{(\texttt{pop}, \varepsilon)\} & b = x \\ \{(\texttt{fail}, x)\} & b \neq x \end{cases} \\
\delta(\texttt{fail}, a, x) & = & \{(\texttt{fail}, x)\} \\
\delta(\texttt{fail}, b, x) & = & \{(\texttt{fail}, x)\}
\end{array}$$

Satz 3.13 *Sei G eine Grammatik in Greibach-Normalform. Dann gibt es einen Kellerautomaten A mit $L(G) = L(A)$, und A akzeptiert L mit leerem Stack.*

Beweis: Sei $G = (\Sigma, V, S, P)$. Wir definieren einen Kellerautomaten mit den folgenden Komponenten:

- $Q = \{q_0\}$: Q besteht also nur aus einem *einziger* Zustand,
- $\Gamma = V$,
- $Z_0 = S$.

Das Programm wird so definiert, dass gilt:

$$S \xrightarrow{*} w_1 \cdots w_n X_1 \cdots X_m \quad \Leftrightarrow \quad \text{Der Automat } A \text{ besitzt nach dem Lesen von } w_1 \cdots w_n \text{ den Kellerinhalt } X_1 \cdots X_m, \text{ wobei } X_1 \text{ zuoberst liegt.}$$

Wir setzen deshalb

$$\delta(q_0, a, y) = \{(q_0, \alpha) \mid y \to a\alpha \in P\}.$$

In Worten: Wenn der Kellerautomat A das Symbol $a \in \Sigma$ liest und $y \to a\alpha$ eine Produktion ist, dann darf

- die auf dem Stack zuoberst liegende Variable y durch α ersetzt werden und
- das nächste Symbol gelesen werden.

Dadurch werden keine ε-Bewegungen benötigt. □

Sind Kellerautomaten mächtiger als kontextfreie Grammatiken?

Satz 3.14 *Die Sprache L werde von einem Kellerautomaten A (mit leerem Stack) akzeptiert. Dann gibt es eine kontextfreie Grammatik G mit*

$$L(G) = L(A).$$

Beweis: (Tripelkonstruktion) Sei $A = (Q, \Sigma, \Gamma, q_0, Z_0, \delta)$ ein Kellerautomat. Angenommen, A hat die Konfiguration

$$(p, u, X_1 \cdots X_m)$$

erreicht und hat bisher $w = w_1 \cdots w_k$ gelesen.

Versuch 1: Dann liegt es nahe, die Grammatik G so zu entwerfen, dass die Ableitung

$$S \xrightarrow{*} w_1 \cdots w_k \; p \; X_1 \cdots X_m$$

möglich ist. *Leider* ist aber die Grammatik als *kontextfrei* zu entwerfen; der Ersetzungsschritt wird also nur vom Zustand p (und nicht auch vom Kellersymbol X_1) abhängen dürfen.

Fazit: Wir müssen Zustand *und* Kellersymbol in einer Variablen zusammenfassen.

Versuch 2:

$$S \xrightarrow{*} w_1 \cdots w_k[p_1, X_1][p_2, X_2] \cdots [p_m, X_m]$$

Was ist p_1, der gegenwärtige Zustand? Was ist p_2? Der *geratene* Zustand, wenn das Symbol X_2, also das Symbol unterhalb von X_1, an die Spitze des Stacks gelangt. Aber wie verifizieren wir, dass p_2 der richtige Zustand ist?

Wir arbeiten mit Tripeln, um uns an den geratenen Zustand zu erinnern!

Versuch 3: Wir entwerfen die Grammatik so, dass

$$S \xrightarrow{*} w_1 \cdots w_k[p_1, X_1, p_2][p_2, X_2, p_3] \cdots [p_m, X_m, p_{m+1}]$$

ableitbar ist.

Die Absicht ist, dass p_i der aktuelle Zustand ist, wenn X_i an die Spitze des Stacks gelangt und dass p_{i+1} der aktuelle Zustand wird, falls das darunterliegende Symbol an die Spitze gelangt. Diese Absicht realisieren wir durch die fogenden Produktionen:

(a) $S \rightarrow [q_0, Z_0, p]$ gehört zur Produktionenmenge für *jedes* $p \in Q$: Wir raten den Endzustand p einer Berechnung.

(b) Für jeden Befehl $(p', \varepsilon) \in \delta(p, a, X)$ fügen wir die Produktion

$$[p, X, p'] \rightarrow a$$

hinzu.

(c) Für jeden Befehl

$$(p_1, \alpha_1 \cdots \alpha_r) \in \delta(p, a, X) \quad \text{ mit } a \in \Sigma \cup \{\varepsilon\}$$

gehören die Produktionen

$$[p, X, q_{r+1}] \rightarrow a[p_1, \alpha_1, p_2][p_2, \alpha_2, p_3] \cdots [p_r, \alpha_r, p_{r+1}]$$

zur Produktionenmenge, wobei alle Kombinationen $(p_2, \ldots, p_{r+1})$ durchlaufen werden.

In einer Linksableitung ist im nächsten Schritt die Variable $[p_1, \alpha_1, p_2]$ zu ersetzen. Wird α_1 durch ein nicht-leeres Wort ersetzt, dann wird der (für die Offenlegung von α_2) geratene Zustand p_2 weiterhin festgehalten. Wird hingegen α_1 mit dem Befehl $(q, \varepsilon) \in \delta(p_1, b, \alpha_1)$ durch das leere Wort ersetzt, dann wird der geratene Zustand p_2 auf die Probe gestellt: Es muss $q = p_2$ gelten. An dieser Stelle setzt also die Verifikation ein! $\square$

Damit haben wir also nachgewiesen, dass die Klasse der kontextfreien Sprachen genau der Menge der Sprachen, die von einem Kellerautomaten mit leerem Stack erkannt werden können, entspricht. Im nächsten Satz machen wir die Aussage, dass die beiden Akzeptanzmodi *„leerer Stack“* bzw. *„akzeptierender Zustand“* gleich mächtig sind.

Satz 3.15 *(a) Sei A_1 ein Kellerautomat, der mit Zuständen akzeptiert. Dann gibt es einen äquivalenten Kellerautomaten A_2, der mit leerem Stack akzeptiert.*

(b) Sei A_1 ein Kellerautomat, der mit leerem Stack akzeptiert. Dann gibt es einen äquivalenten Kellerautomaten A_2, der mit Zuständen akzeptiert.

Beweis: (a) Hier ist die Idee: Wenn A_1 irgendwann einen akzeptierenden Zustand erreicht, dann sollte A_2 die Option haben, den Stack zu leeren. Wenn aber A_1 am Ende seiner Berechnung „zufälliger Weise“ einen leeren Stack geschaffen hat, dann sollte A_2 nicht notwendigerweise akzeptieren.

Konstruktion von A_2:

- Am Anfang ersetze das Kellersymbol Z_0 durch $Z_0'Z_0$ (kein „zufälliges“ Akzeptieren durch leeren Stack).

- A_2 übernimmt das Programm δ_1 von A_1. Wenn ein Zustand in F erreicht wird, dann darf A_2 *nichtdeterministisch* seinen Stack leeren.

(b) Wir möchten, dass A_2 bei leerem Stack (!) noch in einen akzeptierenden Zustand wechseln kann.

Konstruktion von A_2:

- Zu Anfang ersetze Z_0 über eine ε-Bewegung durch $Z_0'Z_0$.

- Wenn das Kellersymbol Z_0' erreicht wird, dann springe in einen akzeptierenden Zustand. □

Damit haben wir also ein Maschinenmodell gefunden, welches genau den kontextfreien Sprachen entspricht. Warum haben wir diese Betrachtungen überhaupt angestellt? Wir hatten mit dem $CYK-Algorithmus$ doch bereits ein Werkzeug, das das Wortproblem einer jeden kontextfreien Sprache in Polynomialzeit löst. Aber die kubische Laufzeit des $CYK - Algorithmus$ bekommt zwar aus *theoretischer* Sicht das Etikett *„effizient“* angeheftet, ist aber für praktische Anwendungen nicht verwendungsfähig. Man überlege sich wie groß der Source-Code eines Programmes werden kann und wie furchterregend dann die Laufzeit wird. Es ist auch festzuhalten, dass der $CYK - Algorithmus$ *immer* in kubischer Zeit läuft. Insbesondere wird das Wortproblem für reguläre Sprachen ebenso nur in kubischer Zeit gelöst.

Wir werden im nächsten Abschnitt deterministische Kellerautomaten, also Kellerautomaten ohne Wahlmöglichkeiten, untersuchen. Wir werden sehen, dass wir durch die Auflage des Determinismus zu einer Einschränkung der Sprachklasse kommen. Wir werden aber auch feststellen, dass die Ausdruckskraft immer noch respektabel ist und dass häufig eine sehr schnelle Compilation möglich ist.

3.4.1 Deterministisch kontextfreie Sprachen

Wir führen nun den deterministischen Kellerautomaten ein.

Definition 3.16

(a) *A heißt ein* <u>*deterministischer Kellerautomat*</u> *(DPDA), wenn*

- *für jedes Tripel* (q, a, z) *(mit* $q \in Q$*,* $a \in \Sigma$ *und* $z \in \Gamma$*)*

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

  gilt und

- *A mit Zuständen akzeptiert.*

(b) *Eine Sprache L heißt* <u>*deterministisch kontextfrei*</u>*, wenn L von einem deterministischen Kellerautomaten erkannt wird.*

Ein deterministischer Kellerautomat erlaubt also höchstens eine Nachfolgekonfiguration, darf aber ε-Schritte durchführen. Beachte, dass die deterministisch kontextfreien Sprachen eine Untermenge der kontextfreien Sprachen sind.

Beispiel 3.8 Wir entwerfen einen deterministischen Kellerautomaten für die Sprache aller legalen Klammerausdrücke, also für die von den Produktionen

$$S \to [S] \,|\, SS \,|\, \varepsilon$$

erzeugte Sprache.

Es ist natürlich $\Sigma = \Big\{[,]\Big\}$. Wir wählen $\Gamma = \Big\{[, Z_0\Big\}$, $Q = \Big\{\texttt{ok}, \texttt{false}\Big\}$ und $q_0 = \texttt{ok}$. Das Programm δ hat die Form

$$\delta\Big(\texttt{ok}, [, [\Big) = \Big\{\Big(\texttt{ok}, [[\Big)\Big\}$$

/* Der Kellerautomat zählt die öffnenden Klammern */

$$\delta\Big(\texttt{ok},], [\Big) = \Big\{\Big(\texttt{ok}, \varepsilon\Big)\Big\}$$

/* Der Automat hat ein Paar zusammengehöriger Klammern gefunden */

$$\delta\Big(\texttt{ok}, [, Z_0\Big) = \Big\{\Big(\texttt{ok}, [Z_0\Big)\Big\}$$

$$\delta\Big(\texttt{ok},], Z_0\Big) = \Big\{\Big(\texttt{false}, Z_0\Big)\Big\}$$

Es wurde keine entsprechende offene Klammer gefunden und deshalb ist `false` der neue Zustand. Im Zustand `false` wird keine Aktion ausgeführt, bis auf das Weiterlesen der Eingabe.

Wenn der Klammerausdruck legal ist, wird für jede öffnende Klammer auch eine schließende Klammer gefunden: Unser Automat hält nur mit leerem Stack.

Wenn unser Automat mit leerem Stack hält, wurde für jede öffnende Klammer auch eine schließende gefunden, und der Klammerausdruck ist legal. Unser Automat sollte also mit leerem Stack akzeptieren.

Wir haben nun die Frage zu beantworten, inwiefern die Auflage des Determinismus zu einer Einschränkung der Sprachklasse führt.

Bemerkung 3.3 Im letzten Abschnitt haben wir einen Kellerautomaten konstruiert, der

$$L = \{ww^{\text{reverse}} | w \in \Sigma^*\}$$

erkennt. Wir haben Nichtdeterminismus eingesetzt, um die Mitte des Wortes zu raten. Man kann aber zeigen, dass ein deterministischer Kellerautomat hier überfordert ist.

Fragen:

1) Warum erlauben wir ε-Bewegungen in der Definition der deterministischen Kellerautomaten?

2) Warum haben wir Akzeptanz durch Zustand gefordert?

Antwort zu 1): Die Sprache $L = \{a^n b^m c^r d^n \,|\, n > 0, m > r, r > 0\}$ kann von einem deterministischen Kellerautomaten mit ε-Bewegung erkannt werden. Ohne ε-Bewegungen erreicht man aber die gespeicherten a's nicht.

An dieser Stelle kommen wir noch einmal auf die beiden uns bekannten Akzeptanzmodi *akzeptierender Zustand* und *leerer Stack* zurück. Im letzten Abschnitt haben wir gesehen, dass die beiden Modi die gleiche Sprachklasse akzeptieren. Für deterministische Kellerautomaten kommen wir hier zu einem negativen Resultat und damit zu einer **Antwort zu 2)**.

Definition 3.17 *Wir definieren*

$$\begin{aligned} \mathcal{L}_D^{\text{Stack}} &= \{L \subseteq \Sigma^* | \exists DPDA \;\; A, \;\; A \text{ akzeptiert } L \text{ mit leerem Stack}\} \\ \mathcal{L}_D^F &= \{L \subseteq \Sigma^* | \exists DPDA \;\; A, \;\; A \text{ akzeptiert } L \text{ mit akzeptierendem Zustand}\}. \end{aligned}$$

Satz 3.18

$$\mathcal{L}_D^{\text{Stack}} \subset \mathcal{L}_D^F$$

Beweis: Im Beweis von Satz 3.15 haben wir aus einem Automaten A_1, der mit leerem Stack akzeptierte, einen Automaten A_2 gebaut, der per Zustand akzeptierte und $L(A_2) = L(A_1)$ erfüllte. Man überzeuge sich, dass diese Konstruktion keinen Nichtdeterminismus hinzugefügt hat. Damit haben wir

$$\mathcal{L}_D^{\text{Stack}} \subseteq \mathcal{L}_D^F$$

nachgewiesen. Wir betrachten nun die Sprache $L = \{0, 01\}$. Zunächst überzeuge man sich, dass L durch einen deterministischen Kellerautomaten akzeptiert werden kann, der mit Zuständen akzeptiert.

Dass diese Sprache nicht von einem deterministischen Kelleratomaten mit leerem Stack akzeptiert werden kann, macht man sich mit einer einfachen Fallunterscheidung klar.

- Schreibt der Automat im Anfangszustand beim Eingabezeichen 0 ein Zeichen auf den Stack, so wird das Eingabewort 0 nicht akzeptiert: Fehler.

- Schreibt der Automat im Anfangszustand beim Eingabezeichen 0 nichts auf den Stack, so bricht die Berechnung nach dem Lesen des ersten Zeichens bereits erfolgreich ab. Auch $00 \notin L$ wird akzeptiert: Fehler.

□

Damit haben wir die Echtheit der Inklusion gezeigt. Akzeptieren per Zustand ist für deterministische Kellerautomaten der mächtigere Akzeptanzmodus.

Bemerkung 3.4 Für deterministisch kontextfreie Sprachen liefert der akzeptierende deterministische Kellerautomat eine schnelle Lösung des Compilerproblems, wenn keine ε-Schritte durchgeführt werden. In diesem Fall erhalten wir eine Lösung in linearer Zeit. Man mache sich aber klar, dass das Fortlassen von ε-Schritten eine wesentliche Einschränkung ist.

Warum ist die Betrachtung von deterministischen kontextfreien Sprachen nun so wichtig?

Satz 3.19 *(a) Jede deterministisch kontextfreie Sprache ist eindeutig.*

(b) $\{ww^{reverse} | w \in \Sigma^*\}$ *ist eindeutig aber nicht deterministisch kontextfrei.*

(c) Das Wortproblem für deterministisch kontextfreie Sprachen kann in Linearzeit gelöst werden.

Beweis: Siehe zum Beispiel das Textbuch von Ingo Wegener. □

Zunächst ist die Eindeutigkeit aus Sicht des Compilerbaus ein immenser Vorteil, da einem Wort mit einer eindeutiger Ableitung leicht eine *Semantik* zugeordnet werden kann. Desweiteren garantiert Teil (c) eine effiziente Lösung des Wortproblems.

3.4.2 Abschlusseigenschaften deterministisch kontextfreier Sprachen

Unter welchen Operationen sind deterministisch kontextfreie Sprachen abgeschlossen? Im Gegensatz zu kontextfreien Sprachen sind deterministisch kontextfreie Sprachen unter Komplementbildung abgeschlossen, aber nicht unter Vereinigung. Als unmittelbare Konsequenz erhalten wir deshalb, dass deterministisch kontextfreie Sprachen eine echte Teilklasse der kontextfreien Sprachen bilden.

Satz 3.20 *(a) Wenn* L *deterministisch kontextfrei ist, dann auch* $\overline{L}$.

(b) Es gibt deterministisch kontextfreie Sprachen L_1 *und* L_2*, so dass der Schnitt* $L_1 \cap L_2$ *nicht einmal kontextfrei ist.*

(c) Die deterministisch kontextfreien Sprachen sind unter Vereinigung nicht abgeschlossen.

(d) Die deterministisch kontextfreien Sprachen bilden eine echte Untermenge der kontextfreien Sprachen.

Beweis: (a) Sei A ein deterministischer Kellerautomat mit $A = (Q, \Sigma, q_0, Z_0, \delta, F)$. Wir „bauen" einen deterministischen Kellerautomaten A', der das Komplement von L akzeptiert.

Beachte, dass A eine Eingabe auch während eventueller ε-Bewegungen akzeptieren kann. Deshalb verwirft A' genau dann, wenn A irgendwann, und zwar nach dem Lesen des letzten Buchstabens und vor dem Versuch den nächsten Buchstaben zu lesen, akzeptiert.

(b) Wir wissen, dass

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

nicht kontextfrei ist. Aber

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

und beide Sprachen sind deterministisch kontextfrei.

(c) Nun zur Vereinigung. Wir betrachten die folgenden Sprachen K_1 und K_2.

$$\begin{aligned} K_1 &= \{a^x b^y c^z \mid x \neq y\} \\ K_2 &= \{a^x b^y c^z \mid y \neq z\} \end{aligned}$$

K_1 und K_2 sind deterministisch kontextfrei, da wir jeweils leicht einen deterministischen Kellerautomaten angeben können. Wie sieht die Vereinigung der Sprachen $K = K_1 \cup K_2$ aus?

$$\begin{aligned} K &= K_1 \cup K_2 \\ &= \{a^x b^y c^z \mid x \neq y \text{ oder } y \neq z\} \end{aligned}$$

Nehmen wir an, K sei deterministisch kontextfrei. Dann wäre wegen Teil (a) auch $\overline{K}$ deterministisch kontextfrei. Wie sieht diese Sprache aus:

$$\overline{K} = \{a^x b^y c^z \mid x = y = z\}$$

Diese Sprache ist aber nicht kontextfrei, womit K nicht deterministisch kontextfrei gewesen sein kann.

(d) Die oben betrachtete Sprache

$$K = \{a^x b^y c^z \mid x \neq y \text{ oder } y \neq z\}$$

ist nicht deterministisch kontextfrei. Da sie aber kontextfrei ist, belegt sie die Echtheit der Inklusion. □

3.5 Entscheidungsprobleme

Nur wenige Entscheidungsprobleme für KFGs sind entscheidbar oder sogar effizient entscheidbar. Das vielleicht wichtigste effizient entscheidbare Problem ist das Wortproblem, das vom Algorithmus von Cocke-Younger-Kasami gelöst wird. Beispiele von weiteren effizient entscheidbaren Problemen sind

- das Leerheitsproblem ($L(G) \stackrel{?}{=} \emptyset$ und
- das Endlichkeitsproblem (ist $L(G)$ endlich?).

Aufgabe 59

(a) Zeige, dass das Leerheitsproblem effizient entscheidbar ist.

Hinweis: Nenne eine Variable X produktiv, wenn es eine Ableitung $X \stackrel{*}{\Rightarrow} w$ für ein Wort w aus Nichtterminalen gibt. Zeige, wie man alle produktiven Variablen effizient bestimmt.

(b) Zeige, dass das Endlichkeitsproblem effizient lösbar ist.

Viele wichtige Entscheidungsprobleme für kontextfreie Sprachen sind hingegen unentscheidbar. Das Postsche Korrespondenzproblem ist die „Ursache allen Übels“.

3.5.1 Das Postsche Korrespondenzproblem

In der Untersuchung der Unentscheidbarkeit von Entscheidungsproblemen für kontextfreie Sprachen und Kellerautomaten spielt das Postsche Korrespondenzproblem (PKP) ein wichtige Rolle. Die Eingabe von PKP besteht aus einem endlichen Alphabet Σ, einer Zahl $k \in \mathbb{N}$ $(k > 0)$ und einer Folge (x_1, y_1), $(x_2, y_2), \ldots, (x_k, y_k)$ von Wortpaaren mit $x_1, y_1, \ldots, x_k, y_k \in \Sigma^+$. Es ist zu entscheiden, ob es ein $n \in \mathbb{N}$ $(n > 0)$ und Indizes $i_1, \ldots, i_n \in \{1, \ldots, k\}$ gibt, so dass $x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n}$ gilt. Mit anderen Worten, es ist zu entscheiden, ob es eine x-Folge gibt, deren Konkatenation mit „ihrer“ Partnerfolge übereinstimmt.

Bei PKP handelt es sich somit um eine Art von Dominospiel, bei dem die Dominosteine nicht wie üblich ein linkes und rechtes Feld, sondern diesmal ein oberes Feld (beschriftet mit x_i) und ein unteres Feld (beschriftet mit y_i) besitzen. Wir müssen die Dominosteine so nebeneinander legen, dass die obere mit der unteren Zeile übereinstimmt.

Beispiel 3.9 Das PKP mit Eingabe $\Sigma = \{0, 1\}$, $k = 3$ und

$$(x_1, y_1) = (1, 111), \qquad (x_2, y_2) = (10111, 10), \qquad (x_3, y_3) = (10, 0).$$

hat eine Lösung mit $n = 4$ und $i_1 = 2$, $i_2 = 1$, $i_3 = 1$, $i_4 = 3$, denn wenn wir die „Dominosteine“ nebeneinander legen erhalten wir

$$\begin{aligned} x_2 x_1 x_1 x_3 &= 10111\ 1\ \ 1\ \ 10 \\ y_2 y_1 y_1 y_3 &= 10\ \ 111\ \ 111\ \ 0 \end{aligned}$$

und die untere stimmt mit der oberen Zeile überein.

Bevor wir zeigen, dass PKP unentscheidbar ist, betrachten wir einige Problemvarianten. Die Eingabe des modifizierten PKP (MPKP) ist identisch zur Eingabe von PKP. Diesmal ist aber zu entscheiden, ob es eine mit x_1 beginnende x-Folge gibt, die mit ihrer Partnerfolge übereinstimmt.

Beispiel 3.10 Betrachtet als Eingabe für das MPKP hat das obige Beispiel keine Lösung. Warum?

PKP_Σ ist eine letzte PKP-Variante: Hier ist das Alphabet nicht mehr Teil der Eingabe, sondern von vorne herein fixiert. Wir reduzieren die universelle Sprache

$$U = \{\ \langle M \rangle w \mid \text{die Turingmaschine } M \text{ akzeptiert Eingabe } w\ \}$$

auf das Postsche Korrespondenzproblem und seine Varianten. Da U unentscheidbar ist, folgt die Unentscheidbarkeit von PKP, MPKP und $\mathrm{PKP}_{\{0,1\}}$.

Satz 3.21 *Es gilt* $U \le \mathrm{MPKP} \le \mathrm{PKP} \le \mathrm{PKP}_{\{0,1\}}$.

Beweisskizze: Wir zeigen zuerst die Reduktion $\mathrm{PKP} \le \mathrm{PKP}_{\{\mathbf{0},\mathbf{1}\}}$. Für ein endliches Alphabet $\Sigma = \{a_1, \ldots, a_m\}$ geben wir zuerst eine Binärkodierung seiner Buchstaben an und zwar kodieren wir a_j durch $h(a_j) = 0\,1^j$ für alle $j \in \{1, \ldots, m\}$. Die Reduktion von PKP auf $\mathrm{PKP}_{\{0,1\}}$ weist einer Eingabe $\Sigma,\ k,\ (x_1, y_1), \ldots, (x_k, y_k)$ für's PKP die folgende Eingabe für's $\mathrm{PKP}_{\{0,1\}}$ zu: k ist unverändert, aber wir benutzen diesmal die binären Wortpaare $(h(x_1), h(y_1)), \ldots, (h(x_k), h(y_k))$.

Warum funktioniert die Reduktion? Für alle $n \in \mathbb{N}$ $(n > 0)$ und alle $i_1, \ldots, i_n \in \{1, \ldots, k\}$ gilt:

$$x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n} \iff h(x_{i_1}) h(x_{i_2}) \cdots h(x_{i_n}) = h(y_{i_1}) h(y_{i_2}) \cdots h(y_{i_n}).$$

Da die Reduktion f berechenbar ist, ist f eine Reduktion von PKP auf $\text{PKP}_{\{0,1\}}$.

Die Reduktion MPKP $\leq$ PKP stellen wir als Übungsaufgabe. Wir kommen also jetzt zum zentralen Ziel, nämlich dem Nachweis der Reduktion $U \leq$ MPKP. **Gesucht** ist eine Transformation f, die jedem Wort $u \in \{0,1\}^*$ eine Eingabe $f(u)$ für's MPKP zuordnet, so dass

$u \in H$, d.h. $u = \langle M \rangle\, w$ für eine TM M, die Eingabe w akzeptiert $\iff$ das MPKP $f(u)$ besitzt eine Lösung.

Betrachten wir zuerst den **leichter Fall**, dass u nicht von der Form $\langle M \rangle\, w$. Dann wählen wir eine Eingabe $f(u)$ für's MPKP, die keine Lösung besitzt, z.B., $\Sigma = \{0,1\}$, $k = 1$, $x_1 = 0$, $y_1 = 1$.

Kümmern wir uns um den **schwierigen Fall**, nämlich $u = \langle M \rangle\, w$ für eine Turingmaschine $M = (\Sigma, Q, \Gamma, \delta, q_0, F)$. Wir nehmen o.B.d.A. an, dass M nur dann in einen akzeptierenden Zustand $q \in F$ wechselt, wenn sie unmittelbar danach anhält. Schließlich fordern wir ebenfalls o.B.d.A, dass M nie den Kopf auf einer Zelle stehen lässt und nie seinen Eingabebereich nach links hin verlässt.

Hier ist die Idee der Reduktion. Wir repräsentieren eine Konfigurationen von M durch Worte über dem Alphabet $\Gamma \dot{\cup} Q$ wie folgt:

> uqv repräsentiert die Situation, bei der M im Zustand q ist, die Bandinschrift uv ist, und der Kopf auf dem ersten Symbol von v steht. Die Startkonfiguration bei Eingabe w wird also durch $q_0 w$ repräsentiert.

Wir geben jetzt eine Eingabe $f(\langle M \rangle\, w)$ für's MPKP an, die aufeinander folgende Konfigurationen von M erzeugt, also die Berechnung von M nachmacht.

- Das Alphabet ist $\Gamma \dot{\cup} Q \dot{\cup} \{\#\}$. Das Symbol # dient als Trennsymbol zwischen den einzelnen Konfigurationen.
- Für ein geeignetes $k \in \mathbb{N}$ wählen wir Wortpaare $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ wie folgt, wobei
$$(x_1, y_1) \text{ mit } x_1 := \# \text{ und } y_1 := \#q_0 w\#$$
das Startpaar ist. Das Wort y_1 beschreibt also die Startkonfiguration, wenn M auf dem leeren Wort rechnet. (Wenn w das leere Wort ist, dann wähle $y_1 = \#q_0\texttt{B}\#$.) Als weitere „Regeln“ verwenden wir
  - für Rechtsbewegungen $\delta(q, a) = (q', a', \texttt{rechts})$ das Paar
  $$(qa, a'q'),$$
  solange $q \neq q_v$,
  - für Linksbewegungen $\delta(q, a) = (q', a', \texttt{links})$ das Paar
  $$(bqa, q'ba')$$
  mit $b \in \Gamma$, solange $q \neq q_v$,
  - die Kopierregeln mit den Paaren
  $$(a, a)$$
  für $a \in \Gamma$

– sowie die Abschlußregeln

$$(\#, \#), \text{ bzw. } (\#, \mathtt{B}\#)$$

für $q \in F$.

Tatsächlich müssen wir noch weitere Regeln aufnehmen, um die folgende Behauptung zu zeigen.

Behauptung: M akzeptiert Eingabe $w \iff f(\langle M \rangle)$ besitzt eine Lösung mit Startpaar (x_1, y_1).

Wie müssen die fehlenden Regeln aussehen? Betrachten wir die Richtung $\Rightarrow$. Unser Ziel ist die Konstruktion übereinstimmender x- und y-Zeilen. Das Startpaar $(\#,\ \#q_0w\#)$ erzwingt dann ein Wort x_2 das mit q_0 beginnt und damit muss die Regel $(q_0w_1, a'q')$ für eine Rechtsbewegung $\delta(q_0, w_1) = (q', a', \mathtt{rechts})$ gewählt werden. Eine solche Regel steht aber nur dann zur Verfügung, wenn M auf w_1 eine Rechtsbewegung im Zustand q_0 durchführt: Das muss M aber tun, denn sonst verlässt es seinen Eingabebereich nach links hin. Also ist zwangsläufig $x_2 = q_0w_1$ und $y_2 = a'q'$.

Die x-Zeile ist jetzt $\#q_0w_1$ und die y-Zeile ist $\#q_0w_1 \cdots w_n\#a'q'w_2$. Wir sind gezwungen ein mit w_2 beginnendes Wort x_3 zu wählen und der Begrenzer $\#$ zwingt uns die Kopierregeln anzuwenden, bis wir die x-Zeile $\#q_0w_1 \cdots w_n$ und die y-Zeile $\#q_0w_1 \cdots w_n\#a'q'w_2 \cdots w_n$ erreicht haben. Auch der nächste Schritt ist erzwungen, denn der Begrenzer muss hinzugefügt werden. (Die Abschlussregel $(\#, \mathtt{B}\#)$ ist zu wählen, wenn $w = w_1$ und M somit im nächsten Schritt das Blanksymbol liest.)

Unsere Regeln gewährleisten also, dass x- und y-Zeile eine Berechnung von M auf Eingabe w simulieren! Wir wissen nach Fallannahme, dass M die Eingabe w akzeptiert und deshalb, wenn wir unsere Konstruktion genügend lange fortsetzen, „endet“ die y-Zeile mit einem Zustand in F. Wir fügen jetzt die Löschregeln

$$(qa, a) \text{ bzw. } (aq, q)$$

für jeden akzeptierenden Zustand q hinzu und können damit (und mit den Kopierregeln) die x-Zeile gegenüber der um die letzte Konfiguration längeren y-Zeile langsam auffüllen. Am Ende haben wir fast Gleichstand erhalten, allerdings ist $y \equiv xq\#$. Wir fügen deshalb noch als letzte Regel

$$(q\#\#, \#)$$

hinzu und haben Gleichstand erreicht.

Da die Umkehrung $\Leftarrow$ mit gleicher Argumentation folgt, ist die Beweisskizze erbracht. □

3.5.2 Unentscheidbare Probleme für kontextfreie Sprachen

Wir zeigen leider, dass viele wichtige Entscheidungsprobleme für kontextfreie Sprachen unentscheidbar sind. Wir reduzieren jedes Mal das Postsche Korrespondenzproblem auf das betreffende Entscheidungsproblem.

Satz 3.22 *Die folgenden Probleme für kontextfreie Grammatiken G, G_1, G_2 und einen regulären Ausdruck R sind unentscheidbar:*

(a) $L(G_1) \cap L(G_2) \neq \emptyset$*?*

(b) Ist die Grammatik G eindeutig?

(c) $L(G) = \Sigma^*$*?*

(d) $L(G_1) = L(G_2)$*? und* $L(G_1) \subseteq L(G_2)$*?*

(e) $L(R) = L(G)$*? und* $L(R) \subseteq L(G)$*?*

Beweis: Gegeben sei ein Postsches Korrespondenzproblem (PKP) über dem Alphabet $\Sigma = \{0,1\}$ mit den Wortpaaren $(x_1, y_1), \ldots, (x_k, y_k)$.

(a) Um das PKP mit Hilfe der Frage „$L(G_1) \cap L(G_2) \neq \emptyset$?“ zu lösen verwenden wir die Sprachen

$$\begin{array}{rcl} L(G_1) & = & \{x_{i_1} \cdots x_{i_n} \$ y_{i_n}^{\text{reverse}} \cdots y_{i_1}^{\text{reverse}} \mid n \in \mathbb{N}\ i_1, \ldots, i_n \in \{1, \ldots, k\}\}, \\ L(G_2) & = & \{w\$w^{\text{reverse}} \mid w \in \{0,1\}^*\ \}. \end{array}$$

Offensichtlich ist das PKP genau dann lösbar, wenn $L(G_1) \cap L(G_2) \neq \emptyset$ gilt. Beachte, dass $L(G_1)$ kontextfrei ist: Benutze die Produktionen

$$S \ \rightarrow \ x_i\, S\, y_i^{\text{reverse}} \mid x_i\, \$\, y_i^{\text{reverse}} \quad \text{für } i = 1, \ldots, k.$$

(b) Wir verwenden das Alphabet $\Sigma = \{0, 1, \overline{1}, \overline{2}, \ldots, \overline{k}\}$ und wählen die kontextfreie Grammatik G mit den Produktionen

$$\begin{array}{rcl} S & \rightarrow & A \mid B \\ A & \rightarrow & x_1 A \overline{1} \mid x_2 A \overline{2} \mid \ \cdots \ \mid x_k A \overline{k} \mid \varepsilon \\ B & \rightarrow & y_1 B \overline{1} \mid y_2 A \overline{2} \mid \ \cdots \ \mid y_k A \overline{k} \mid \varepsilon. \end{array}$$

G ist genau dann eindeutig, wenn das PKP keine Lösung besitzt.

(c) Wir definieren die Sprache

$$L_1 = \{w\$w^{\text{reverse}} \mid w \in \{0,1\}^*\ \},$$

wobei aber das Zeichen # an beliebiger Stelle in beliebiger Anzahl vorkommen kann, sowie die Sprache

$$L_2 = \{x_{i_1} \# \cdots \# x_{i_n}\, \$\, y_{i_n}^{\text{reverse}} \# \cdots \# y_{i_1}^{\text{reverse}} \mid n \in \mathbb{N}\ i_1, \ldots, i_n \in \{1, \ldots, k\}\}.$$

L_1 und L_2 sind deterministisch kontextfrei. Also sind sowohl $\{0,1,\#,\$\}^* \setminus L_1$ wie auch $\{0,1,\#,\$\}^* \setminus L_2$ kontextfrei. Dann ist aber auch

$$\begin{array}{rcl} L & = & (\{0,1,\#,\$\}^* \setminus L_1) \cup (\{0,1,\#,\$\}^* \setminus L_2) \\ & = & \{0,1,\#,\$\}^* \setminus (L(G_1) \cap L(G_2)) \end{array}$$

kontexfrei. Also ist das PKP genau dann unlösbar, wenn $L = \{0,1,\#,\$\}^*$.

(d) und **(e)** sind unmittelbare Konsequenz von (c). □

3.5.3 Das Äquivalenzproblem für Finite State Transducer

Die Unentscheidbarkeit des Postschen Korrespondenzproblems hat viele Konsequenzen für die Unentscheidbarkeit von Problemen im Bereich der Formalen Sprachen. Wir geben eine erste solche Konsequenz für *Finite State Transducer* (FST) an: Ein FST $A = (Q, \Sigma, \Gamma, I, F, \delta)$ ist wie ein nichtdeterministischer endlicher Automat mit Ausgabe aufgebaut. Insbesondere ist

- Q die Zustandsmenge, $I \subseteq Q$ die Menge der Anfangszustände und F die Menge der akzeptierenden Zustände,

- Σ ist das Eingabealphabet und Γ das Ausgabealphabet.
- Die Überführungsrelation δ hat die Form

$$\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\} \times \Gamma \cup \{\varepsilon\}) \times Q.$$

  Wenn sich die Maschine A im Zustand q befindet und den Buchstaben $a \in \Sigma$ liest, dann darf A genau dann in den Zustand q' wechseln und den Buchstaben a' drucken, wenn $(q, a, a', q') \in \delta$.

Wir sagen, dass A unter Eingabe $x \in \Sigma^*$ die Ausgabe $y \in \Gamma^*$ produzieren kann, wenn es einen Anfangszustand $i \in I$ gibt, so dass eine in Zustand i beginnende Berechnung für Eingabe x die Ausgabe y produziert.

Die Äquivalenz von NFA's ist eine durchaus schwierige Aufgabe, die aber entscheidbar ist: Überführe die nfa's in dfa's mit Hilfe der Potenzmengenkonstruktion und überprüfe die Äquivalenz der dfa's. Die Äquivalenz von FST's hingegen ist überraschender Weise (?!) unentscheidbar.

Korollar 3.23 *Die Frage, ob zwei gegebene Finite State Transducer dieselbe Sprache erkennen, ist nicht entscheidbar.*

Beweis: Wir reduzieren $\text{PKP}_{\{0,1\}}$ auf das Äquivalenzproblem für Finite State Transducer.

Seien k und die Paare $(x_1, y_1), \ldots, (x_k, y_k)$ eine Eingabe für $\text{PKP}_{\{0,1\}}$. Für das Eingabealphabet $\Sigma = \{1, \ldots, k\}$ und das Ausgabealphabet $\Gamma = \{0, 1\}$ bauen wir zuerst einen recht trivialen FST A_1, der für jede Eingabe $\xi \in \Sigma^*$ jede Ausgabe $\eta \in \Gamma^*$ produzieren kann. Der FST A_2 ist ebenfalls sehr einfach: Für eine Eingabe $i_1 \cdots i_n \in \Sigma^*$ produziert A_2 jede mögliche Ausgabe $w \in \Gamma^*$, falls $w \neq x_{i_1} \cdots x_{i_n}$ oder $w \neq y_{i_1} \cdots y_{i_n}$.

Angenommen, es gibt eine Lösung für $\text{PKP}_{\{0,1\}}$ und die Eingabe $(x_1, y_1), \ldots, (x_k, y_k)$. Dann gibt es $i_1, \ldots, i_n$ mit $x_1 \cdots x_k = y_1 \cdots y_k$. Dann wird aber A_2 nicht die Ausgabe $x_{i_1} \cdots x_{i_n}$ produzieren und damit unterscheidet sich A_1 von A_2. Ist $\text{PKP}_{\{0,1\}}$ für Eingabe $(x_1, y_1), \ldots, (x_k, y_k)$ hingegen nicht lösbar, dann sind A_1 und A_2 äquivalent. □

Aufgabe 60
Zeige: Die Frage, ob zwei gegebene Kellerautomaten dieselbe Sprache akzeptieren, ist nicht entscheidbar.

3.6 Zusammenfassung

Wir haben gesehen, dass kontextfreien Sprachen eine wesentlich größere Sprachklasse abdecken als die regulären Sprachen. Allerdings haben das Pumping-Lemma bzw. das mächtigere Lemma von Ogden auch die Grenzen der Ausdrucksstärke kontextfreier Sprachen gezeigt. Das Konzept des Ableitungsbaums hat sich in diesen Untersuchungen als besonders hilfreich erwiesen.

Wir haben die *Chomsky-Normalform* kontextfreier Sprachen eingeführt und gezeigt, dass wir mit Hilfe der dynamischen Programmierung *der (CYK-Algorithmus)* das Wortproblem in Zeit

$$O(|\text{Produktionen}| \cdot |w|^3)$$

lösen können. Für praktische Anwendungen ist das Konzept der kontextfreien Sprachen dennoch etwas zu weit gefaßt. Kubische Laufzeit ist ab einer gewissen Sourcecodelänge nicht mehr akzeptabel.

Als Konsequenz haben wir die kontextfreien Sprachen noch weiter eingeschränkt und kamen zu den *deterministisch kontextfreien* Sprachen, die wir mit Hilfe *deterministischer Kelleratomaten* eingeführt haben. Nebenbei haben wir beobachtet, dass nichtdeterministische Kellerautomaten genau die kontextfreien Sprachen akzeptieren. Die wichtigen Eigenschaften der Klasse deterministisch kontextfreier Sprachen sind einerseits ihre Eindeutigkeit und andererseits die Lösung des jeweiligen Wortproblems in Linearzeit.

Mit Hilfe des Postschen Korrespondenzproblems (PKP) kann die Unentscheidbarkeit vieler Entscheidungsprobleme für kontextfreien Sprachen und Kellerautomaten gezeigt werden. Wir haben die Unentscheidbarkeit des Äquivalenzproblems für kontextfreie Grammatiken und Finite State Transducer mit Hilfe der Unentscheidbarkeit von PKP nachgewiesen.

Kapitel 4

XML und reguläre Baumgrammatiken*

Im Compilerproblem haben wir uns die Aufgabe gestellt, die syntaktische Korrektheit eines Programms P zu überprüfen. Dazu haben wir uns P als einen String, nämlich als Konkatenation aller Anweisungen von P vorgestellt. Im Typechecking Problem für XML-Dokumente sehen wir uns mit einer neuen Situation konfrontiert:

> Ein XML-Dokument D ist hierarisch strukturiert. Eine Interpretation von D als eine Folge von Buchstaben unterdrückt die Zusatzinformation der hierarischen Strukturierung und ist deshalb nicht angemessen. Vielmehr stellt es sich heraus, dass wir D als einen Baum auffassen sollten, dessen Knoten mit Worten beschriftet sind.

Wir werden deshalb Grammatiken betrachten, die beschriftete Bäume erzeugen und untersuchen Baumsprachen, also Sprachen, die nicht mehr aus Strings, sondern aus beschrifteten Bäumen bestehen. Wir schließen die Betrachtung von XML mit den Sprachen XPath und XQuery, die für die Auswertung von XML-Dokumenten vor Allem in Hinblick auf Datenbankanwendungen eingesetzt werden.

4.1 XML

Wir geben eine kurze Beschreibung von XML (extensible Markup Language oder in Deutsch: erweiterbare Auszeichnungssprache[1]). XML ist ein vom World Wide Web Konsortium W3C verabschiedeter Standard für einen plattform-unabhängigen Austausch von Daten zwischen Rechnern. Die Regeln von XML unterstützen eine hierarische Strukturierung, also eine Baumstrukturierung der Daten.

Wir geben zuerst einen kurzen Überblick über die Syntax von XML-Dokumenten. Ein XML-Dokument entspricht einem Baum aus Elementen. Elemente mit Inhalt besitzen eine Start- und Endkennung, die jeweils mit dem Namen des Elements beschriftet ist:

```
<element-name>
   element-beschreibung
</element-name>
```

[1] HTML und LaTex sind Beispiele „konventioneller“ Auszeichungssprachen.

Elemente ohne Inhalt werden durch `<element-name element-beschreibung />` spezifiziert. Elemente mit Inhalt besitzen Kinder-Elemente und Attribute, Elemente ohne Inhalt besitzen nur Attribute. Jedes XML-Dokument besitzt eine Baumstruktur. Dazu fordert die Syntax von XML, dass ein XML-Dokument

- genau ein Wurzelelement besitzt und
- dass die Kennung eines Elements vor der Endkennung des Elternelements oder der Startkennung eines Geschwisterelements zu schließen ist.

Beispiel 4.1 Ein XML-Rezept.

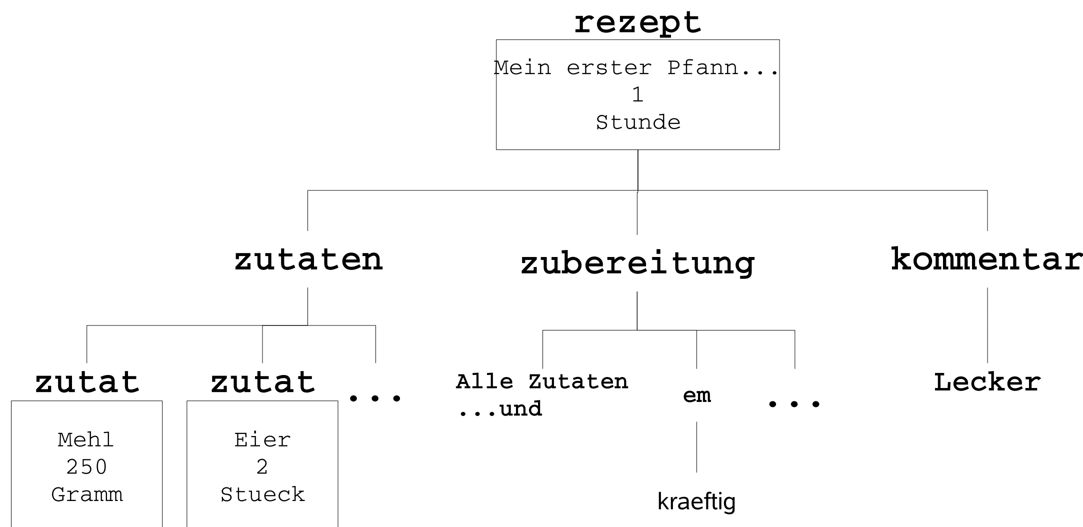
```
<rezept name="Mein erster Pfannkuchen" benoetigte-zeit="1" einheit="Stunde">
   <zutaten>
            <zutat     name="Mehl"     wieviel="250"  einheit="Gramm"/>
            <zutat     name="Eier"     wieviel="2"    einheit="Stueck"/>
            <zutat     name="Zucker"   wieviel="1"    einheit="Essloeffel"/>
            <zutat     name="Milch"    wieviel="500"  einheit="Milliliter"/>
            <zutat     name="Rapsoel"  wieviel="5"    einheit="Gramm"/>
   </zutaten>
   <zubereitung>
            Alle Zutaten in eine Schuessel geben und
            <em>kraeftig</em> umruehren.  Dann Rapsoel
            in der Pfanne erhitzen, den Teig in die Pfanne
            geben und <em>rechtzeitig</em> herausnehmen.
  </zubereitung>
  <kommentar>
            Lecker
  </kommentar>
</rezept>
```

Die erste Annotation `<rezept>...</rezept>` definiert das Wurzelelement mit Startkennung (engl: start tag) `<rezept>` und Endkennung (engl: end tag) `</rezept>`. Der von `<rezept>` und `</rezept>` eingeschlossene Teil definiert den Inhalt des Elements, also in unserem Fall

- die in der Reihenfolge `zutaten, zubereitung, kommentar` geordneten „regulären" Kinder sowie
- die Attribute `name, benoetigte-zeit` und `einheit`, die als ungeordnete, „irreguläre" Kinder des Elements auftreten.

  Ein Attribut kommt stets im Inhalt eines Elements vor, besitzt keine Kinder und nimmt Werte an.

Das Element `rezept` definiert also einen XML-Baum mit Wurzel `rezept` und den drei geordneten Kindern `zutaten, zubereitung,kommentar`. Der Inhalt von `zutaten` besteht aus den fünf `zutat` Elementen, die ihrerseits drei Attributkinder, nämlich `name`, `wieviel` und `einheit` besitzen. Das Element `zubereitung` besitzt fünf Kinder, ein Textknoten gefolgt von dem Element `em`, einem Textknoten, einem weiteren Vorkommen von `em` und einem abschließendem Textknoten. Schließlich hat `kommentar` einen Textknoten als Kind.



Die Start-Ende Kennungen eines XML-Dokuments legen also den zu einem XML-Dokument gehörenden Baum eindeutig fest und dieser Baum ist sehr schnell bestimmbar.

Definition 4.1 *Ein XML-Baum* (W, E, λ, ν) *ist ein geordneter Baum mit Knotenmenge* W *und Kantenmenge* E*. Innere Knoten entsprechen Elementen, Blätter entsprechen Attributen.*

Die Funktion λ *weist einem Textknoten die Markierung „Text“, jedem anderen Knoten seine Kennung, bzw seinen Attributnamen zu. Die Funktion* ν *weist Textknoten den zugehörigen Text und Attributen Werte zu.*

Wir unterscheiden also zwischen der Beschriftung λ, die Textknoten identifiziert, bzw. den Namen (für ein Element oder ein Attribut) zuweist und der Beschriftung ν, die Werte zuweist. Im Folgenden werden wir uns nur auf die Graphstruktur eines XML-Baums und die Namensgebung λ konzentrieren, die Wertezuweisung werden wir also unterschlagen.

Jedes XML-Dokument D wird also durch einen XML-Baum B repräsentiert. Wie erhalten wir D, wenn nur B gegeben ist? Wir drucken die Knoten von B in einem Präorderdurchlauf von B aus und weisen Werte mit Hilfe der Funktion ν zu!

Anwendungsspezifische XML-Sprachen wie

- RSS (Veröffentlichung von Änderungen auf Webseiten),
- XHTML (Neuformulierung von HTML 4 in XML 1.0),
- SVG –scalable vector graphics– (Beschreibungssprache für zweidimensionale Vektorgrafiken),
- MathML (Darstellung mathematischer Formeln),
- GraphML (Beschreibung und Darstellung von Graphen)

schränken die XML-Syntax ein. Jede dieser Sprachen unterscheidet zwischen zwei Klassen von XML-Dokumenten, nämlich den sinnvollen, für die das XML-Dokument syntaktisch korekt ist und den Rest. XML Schema Sprachen geben die Möglichkeit, eine Syntax für XML-Dokumente zu beschreiben. XML-Parser (oder XML-Prozessoren) überprüfen dann die syntaktische Korrektheit eines vorgegebenen XML-Dokuments. Wir konzentrieren uns im Folgenden auf grammatik-basierte Schema Sprachen (wie DTD, XML Schema, RELAX NG und XSLT), also auf die am weitesten verbreitete Klasse von Schema Sprachen.

4.2 DTDs

Eine Syntaxbeschreibung durch DTDs hat die Form

```
<!DOCTYPE Name-der-Syntax
 [ eine Liste von Elementdefinitionen ]>
```

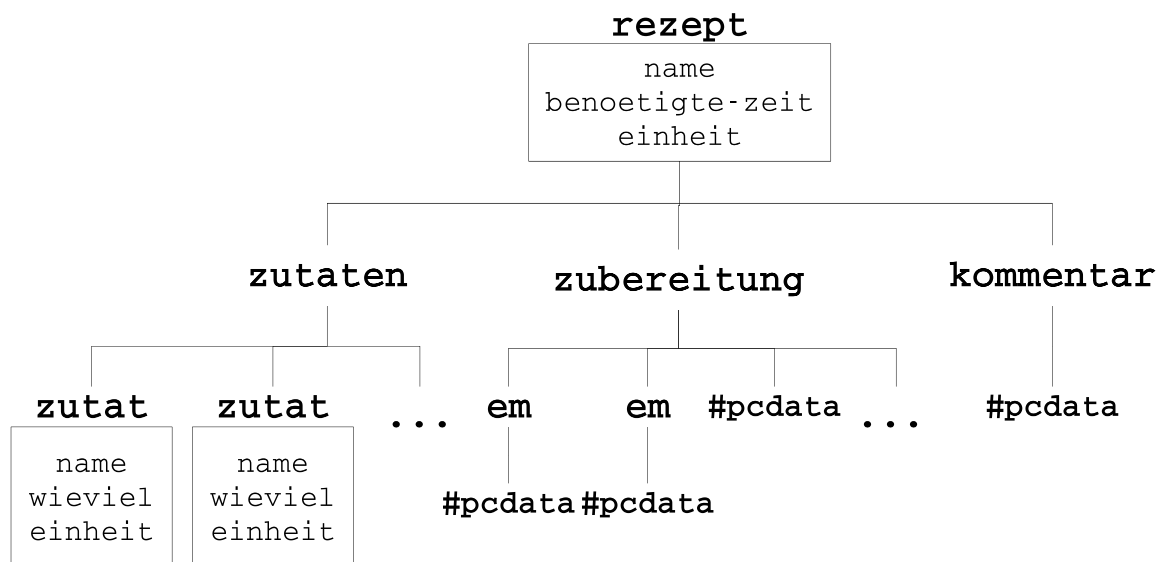
Der Typ eines Elements wird dann durch die Beschreibung

```
<!ELEMENT Name-des-Elements ( Definition-des-Elements )>
```

festgelegt. Eine DTD (Document Type Definition) legt fest, welche Wahlmöglichkeiten für die Kinder eines jeden Elements bestehen. Diese „Kinderregeln“ weisen dazu dem Element v einen regulären Ausdruck in den „Kinder-Elementen“ von v zu.

Beispiel 4.2 Eine DTD für XML-Rezepte könnte zum Beispiel das folgende Aussehen haben:

```
<!ELEMENT rezept          (zutaten, zubereitung, kommentar?)>
<!ELEMENT zutaten         (zutat*)>
<!ELEMENT zutat           >
<!ELEMENT zubereitung     ((#PCDATA | em )*)>
<!ELEMENT em              (#PCDATA)>
<!ELEMENT kommentar       (#PCDATA)>
<!ATTLIST rezept          name               CDATA #REQUIRED
                          benoetigte-zeit    CDATA #REQUIRED
                          einheit            CDATA #REQUIRED>
<!ATTLIST zutat           name               CDATA #REQUIRED
                          wieviel            CDATA #REQUIRED
                          einheit            CDATA #REQUIRED>
```



In den regulären Ausdrücken steht ein Komma für die Konkatenation, der vertikale Strich für die Vereinigung, der Stern für den Kleene-Operator und das Fragezeichen für die optionale Wahl. Zusätzlich wird `+` für ein mindestens einmaliges Auftreten verwandt.

Das Schlüsselwort `#PCDATA` steht für parsed character data und gibt an, dass der jeweilige Textknoten Stringwerte annimmt. Die Deklarationen von Attributen geschieht durch die `<!ATTLIST ... >` Kennung; die Markierung `#REQUIRED` gibt an, dass das Attribut auftreten muss.

Die DTD beschreibt also die Klasse zulässiger Bäume mit Wurzel `rezpt`. Zum Beispiel legen wir fest, dass `rezept` das linke Kind `zutaten` und darauffolgend das Kind `zubereitung` besitzt; möglicherweise hat `rezept` auch noch `kommentar` als rechtestes Kind. `zutaten` hat unbeschränkt viele Kinder vom Typ `zutat`, wobei in `zutat` die Attribute `name, wieviel, einheit` auftreten.

Ein Element darf durchaus mehrfach im Baum vorkommen. Bei mehrmaligem Auftreten im Baum gilt aber stets die eine in der DTD beschriebene Deklaration, verschiedene, vom Kontext-abhängige Deklarationen sind also verboten.

Das Wortproblem für nichtdeterministische Automaten kann in Zeit $O(|w| \cdot |Q|^2)$ für ein Eingabewort w gelöst werden. Da ein regulärer Ausdruck der Länge L in einen nichtdeterministischen Automaten mit höchstens L Zuständen umgebaut werden kann, ist also auch das Wortproblem für reguläre Ausdrücke effizient lösbar. Um aber eine noch schnellere Berechnung zu erlauben, werden in DTDs nur *deterministische reguläre Ausdrücke* zugelassen.

Definition 4.2 *Sei R ein regulärer Ausdruck über dem Alphabet Σ. Wir ersetzen R durch den regulären Ausdruck R', indem für alle i und alle Buchstaben $a \in \Sigma$ das ite Vorkommen von a durch den neuen Buchstaben a_i ersetzt wird.*

R ist deterministisch, wenn es keinen Buchstaben $b \in \Sigma$ und keine Worte w, v, v' gibt, so dass die beiden Worte wb_iv und wb_jv' mit $i \neq j$ den Ausdruck R' erfüllen.

Der reguläre Ausdruck $R = ab + ac$ ist also nicht deterministisch, denn sowohl a_1b wie auch a_2c erfüllen den Ausdruck $R' = a_1b_1 + a_2c_1$. Beachte aber, dass der äquivalente Ausdruck $R = a(b+c)$ sehr wohl deterministisch ist, denn jeder Buchstabe taucht genau einmal auf.

Das Wortproblem für einen deterministischen Ausdruck R lässt sich in einem links-nach-rechts Durchlauf des Worts Eingabeworts u lösen, da für jeden Buchstaben b nur ein Vorkommen von b in R "passt". Wenn wir das Wortproblem für das Eingabewort aaa und den Ausdruck $(a+b)^*a$ lösen möchten, dann wissen wir nicht, welches Vorkommen von a in $(a+b)^*a$ für das erste a in aaa passt. Der mit $(a+b)^*a$ äquivalente (und deterministische) Ausdruck $b^*a(b^*a)^*$ stellt kein Problem dar. (Warum?)

Aufgabe 61
Zeige, dass $(a+b)^*a$ und $b^*a(b^*a)^*$ äquivalent sind und dass $b^*a(b^*a)^*$ ein deterministischer Ausdruck ist.

Aufgabe 62
Zeige, dass das Wortproblem für deterministische reguläre Ausrücke in einem links-nach-rechts Durchlauf des Eingabeworts gelöst werden kann.

Wenn wir die Eigenheiten der DTD-Syntax ausblenden und auch die Deklaration der Attribute vernachlässigen, werden wir auf den folgenden Kern der Deklaration geführt.

```
Rezept      -> rezept(Zutaten, Zubereitung, Kommentar?)
Zutaten     -> zutaten(Zutat*)
Zutat       -> zutat()
Zubereitung -> zubereitung((#PCDATA | Em)*)
Em          -> em(#PCDATA)
Kommentar   -> kommentar(#PCDATA)
#PCDATA     -> #pcdata
```

Wir können diese Deklarationen auch als Produktionen einer *erweiterten* kontextfreien Grammatik auffassen: Die Variablen der Grammatik entsprechen den Elementen, die Produktionen den regulären Ausdrücken. Beachte, dass wir Elementnamen gross schreiben, wenn wir das jeweilige Element als Variable auffassen. Die Kleinschreibung deutet die Sichtweise als „Buchstabe" an: Start- und Endkennung mit dem jeweiligen Namen sind einzufügen. Unsere Produktionen sind also stets von der Form

$$X \to a(R), \tag{4.1}$$

wobei X eine Variable, a ein Buchstabe und R ein regulärer Ausdruck ist, der nur aus Variablen besteht.

Genauso wie in diesem Beispiel können wir auch *beliebige* DTD-Deklarationen als eine erweiterte[2] kontextfreie Grammatik G auffassen und erhalten als Konsequenz:

> Wenn wir Attribute und Text-Elemente vernachlässigen, dann entsprechen XML-Bäume genau den Ableitungsbäumen von G.

Das Wortproblem (oder das Typchecking Problem) für einen vorgegebenen DTD-Typ ist deshalb wesentlich einfacher als für allgemeine kontextfreie Grammatiken, denn wir müssen ja nur überprüfen, ob ein vorgegebener Ableitungsbaum legal ist.

Ein Parser für DTDs

> Das XML-Dokument D liege mit einer DTD-Deklaration vor. Typechecking wird durch einzigen links-nach-rechts Durchlauf von D durchgeführt:
>
> (1) Wenn eine Startkennung mit Namen a angetroffen wird, dann bestimme die eine Produktion $X \to aR$, die a erzeugen kann.
>
> (2) Wenn die entsprechende Endkennung gelesen wird, dann überprüfe, ob die den Kindern zugeordnete Buchstabenfolge den regulären Ausdruck R erfüllt. Ist dies nicht der Fall, dann brich mit einer Fehlermeldung ab.
> /* Hier zahlt sich die DTD-Beschränkung auf deterministische Ausdrücke aus, da das Wortproblem für R ebenfalls in einem links-nach-rechts Durchlauf lösbar ist. */

Typechecking geschieht also blitzschnell und benötigt noch nicht einmal eine explizite Bestimmung des XML-Baums. Man sagt auch, dass der Parser im „Event Modell" arbeitet, wobei jede Start- und Endkennung ein Event, also ein Ereignis darstellt. Wir können uns natürlich die Arbeitsweise des Parsers auch auf dem zugehörigen XML-Baum vorstellen, wenn wir uns erinnern, dass ein XML-Dokument aus seinem Baum durch einen Präorderdurchlauf entsteht: Der Parser arbeitet den Baum also nach der Präorderreihenfolge ab.

4.3 Reguläre Baumsprachen

Wie groß ist die Beschreibungskraft von DTDs, beziehungsweise welche anderen vernünftigen Beschreibungsmechanismen stehen uns noch zur Verfügung? Um diese Fragen zu beantworten, stellen wir jetzt reguläre Baumgrammatiken vor, die beschriftete Bäume und insbesondere XML-Bäume erzeugen.

[2]Natürlich könnten wir auch konventionelle kontextfreie Grammatiken verwenden, müssten dann aber die regulären Ausdrücke durch mehrere konventionelle Produktionen ersetzen, ein kleiner Schönheitsfehler.

Definition 4.3 *Eine reguläre Baumgrammatik (V, Σ, S, P) besteht aus*

- *der endlichen Menge V der Variablen,*
- *dem Alphabet Σ,*
- *dem Startsymbol $S \in V$*
- *und einer endlichen Menge P von Produktionen.*

Jede Produktion in P ist von der Form

$$X \to aR,$$

wobei $X \in V$ eine Variable, $a \in \Sigma$ ein Buchstabe und R ein regulärer Ausdruck ist, der nur aus Variablen besteht. Wir sagen auch, dass R die Kinderstruktur von X beschreibt.

Beachte, dass wir annehmen können, dass es zu jeder Variablen $X \in V$ und zu jedem Buchstaben $a \in \Sigma$ höchstens eine Produktion $X \to aR$ gibt: Wir können ja zwei Produktionen $X \to aR_1$ und $X \to aR_2$ zu der einen Produktion $X \to a(R_1 + R_2)$ vereinigen.

Eine erste Beobachtung zeigt, dass wir eine DTD Deklaration als eine reguläre Baumgrammatik auffassen können, wenn wir statt der Produktionen $X \to a(R)$ in (4.1) die Produktionen $X \to aR$ verwenden. Reguläre Baumgrammatiken stimmen also mit kontextfreien Grammatiken in Greibach Normalform überein, der Unterschied ist natürlich, dass diesmal Bäume und nicht Strings erzeugt werden:

> Der Buchstabe a ist der Name des zugehörigen Knotens und R beschreibt die Kinderstruktur des Knotens.

Definition 4.4 *(a) Sei $G = (V, \Sigma, S, P)$ eine reguläre Baumgrammatik und sei (W, E, λ) ein beschrifteter Baum: Es ist also (W, E) ein geordneter Baum und $\lambda : W \to \Sigma$ ist seine Beschriftung. Wir sagen, dass G den Baum (W, E, λ) genau dann erzeugt, wenn es eine Funktion $\lambda' : W \to V$ mit den folgenden Eigenschaften gibt:*

- *Für die Wurzel w des Baums gilt $\lambda'(w) = S$.*
- *Für jeden Knoten $u \in W$ mit den (geordnet aufgelisteten) Kindern $u_1, \ldots, u_k$ gibt es eine Produktion $X \to aR$, so dass*
  - *$\lambda'(u) = X$,*
  - *$\lambda(u) = a$ und*
  - *$\lambda'(u_1)\lambda'(u_2) \cdots \lambda'(u_k)$ erfüllt den Ausdruck R.*

Die von G erzeugte Baumsprache $L(G)$ besteht aus allen von G erzeugten Bäumen.

(b) Eine Sprache L ist genau dann eine reguläre Baumsprache, wenn $L = L(G)$ für eine reguläre Baumgrammatik gilt.

Wir verwenden im Folgenden Großschreibung für Variablen und Kleinschreibung für Terminale. Um die Lesbarkeit zu erleichtern, benutzen wir runde Klammern, um reguläre Ausdrücke von „ihrem“ Terminal abzugrenzen.

Beispiel 4.3 Wir konstruieren eine reguläre Baumgrammatik, die alle arithmetischen Ausdrucksbäume erzeugt. Wir verwenden nur eine Variable `Ausdruck`, die natürlich auch gleichzeitig das Startsymbol ist. Das Alphabet ist $\Sigma = \{+, -, *, /, \texttt{variable}\}$, und wir verwenden die Produktionen

```
Ausdruck -> + ( Ausdruck , Ausdruck+)
Ausdruck -> - ( Ausdruck , Ausdruck )
Ausdruck -> * ( Ausdruck , Ausdruck+)
Ausdruck -> / ( Ausdruck , Ausdruck )
Ausdruck -> variable()
```

(Für eine Variable X bezeichnet $X+$ beliebige Wiederholungen von X, wobei X mindestens einmal vorkommen muss.)

Beispiel 4.4 Wir sagen, dass ein Binärbaum B ein Und/Oder Baum ist,

- wenn die Blätter mit entweder `0` oder `1`,
- die inneren Knoten mit entweder `and` oder `or` beschriftet sind
- und wenn B als Und/Oder Schaltung aufgefasst, den Wert 1 berechnet.

Wir zeigen, dass die Sprache aller Und/Oder Bäume eine reguläre Baumsprache ist. Wir verwenden die Variablenmenge $V = \{\texttt{Null}, \texttt{Eins}\}$, das Alphabet $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{and}, \texttt{or}\}$, das Startsymbol $S = \texttt{Eins}$ und die Produktionen

```
Null -> or  (  Null , Null )
Null -> and ( (Eins , Null ) | ( Null , Eins) | (Null , Null) )
Null -> 0()
Eins -> or  ( (Eins , Null ) | ( Null , Eins) | (Eins , Eins) )
Eins -> and (  Eins , Eins )
Eins -> 1()
```

Wie sehen DTD-Deklarationen aus, wenn wir sie als reguläre Baumgrammatiken auffassen?

Definition 4.5 *Sei $G = (V, \Sigma, S, P)$ eine reguläre Baumgrammatik.*

(a) Wir sagen, dass zwei Variablen $X, Y \in V$ im Buchstaben $a \in \Sigma$ kollidieren, falls es zwei Produktionen $X \to aR_1, Y \to aR_2$ mit $R_1 \neq R_2$ in P gibt.

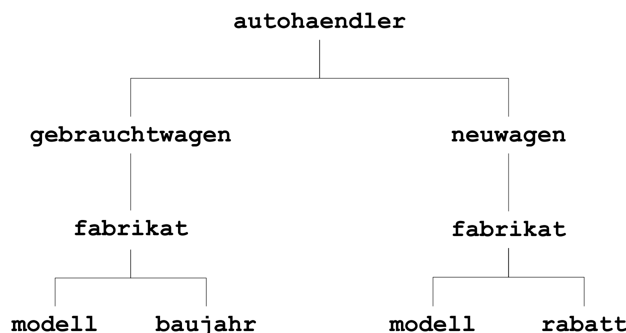
(b) G heißt lokal, wenn keine zwei Variablen in einem Buchstaben kollidieren.

(c) Eine reguläre Baumsprache L heißt lokal, wenn L von einer lokalen Baumgrammatik erzeugt wird.

Wenn wir die DTD-Beschränkung auf deterministische reguläre Ausdrücke aufheben, dann stimmen DTD-Deklarationen in ihrer Ausdruckskraft mit lokalen Baumgrammatiken überein, denn in einer DTD-Deklaration gibt es für jeden Elementnamen a genau eine Produktion, die den Buchstaben a erzeugen kann, also von der Form $X \to aR$.

Beispiel 4.5 Die folgende Baumsprache, die nur aus dem Baum B besteht, ist nicht durch DTD-Deklarationen erzeugbar:

B hat die Wurzel `autohaendler`. Der linke Teilbaum hat die Wurzel `gebrauchtwagen` mit dem einzigem Kind `fabrikat` und den beiden Enkelkindern `modell` und `baujahr`. Der rechte Teilbaum hat die Wurzel `neuwagen` mit dem einzigen Kind `fabrikat` und den beiden Enkelkindern `modell` und `rabatt`.



Aufgabe 63
Zeige, dass es keine lokale Baumgrammatik gibt, die nur den Baum B erzeugt.

Damit ist die „Ausdruckskraft" regulärer Baumgrammatiken sehr viel größer als die Ausdruckskraft von DTD-Deklarationen. Diese geringe Ausdruckskraft ist ärgerlich, weil zum Beispiel Kontext-abhängige Deklarationen von Elementen wie in Beispiel 4.5 ausgeschlossen sind.

4.3.1 Baumsprachen mit eindeutigen Typen

Wir kommen jetzt zu XML Schema, einer Schema Sprache, die gegenwärtig am weitesten verbreitet ist. (XML Schema ist durchaus umstritten, was zum Teil auf ihre extrem aufwändige Beschreibung zurückzuführen ist.) Die Ausdruckskraft von XML Schema im Vergleich zu DTDs nimmt zu. Von der Perspektive der regulären Baumsprachen aus gesehen, lassen sich XML Schema Deklarationen als *Baumgrammatiken mit eindeutigen Typen* (single type tree grammar) auffassen.

Definition 4.6 *$G = (V, \Sigma, S, P)$ sei eine reguläre Baumgrammatik.*

(a) Wir sagen, dass G eindeutige Typen hat genau dann, wenn für jede Produktion $X \to aR$ in P keine zwei in R auftretenden Variablen in einem Buchstaben aus Σ kollidieren.

(b) Eine reguläre Baumsprache L hat eindeutige Typen, wenn L von einer Baumgrammatik mit eindeutigen Typen erzeugt wird.

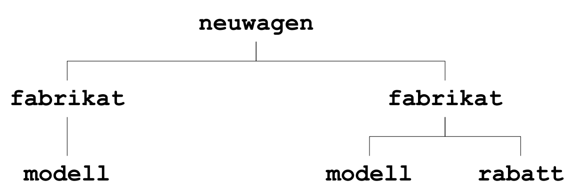
Wir zeigen jetzt, dass der Baum aus Beispiel 4.5 durch eine Baumgrammatik mit eindeutigen Typen erzeugt werden kann: XML Schema ist also mächtiger als DTDs.

Wir verwenden die Produktionen

```
Autohaendler   -> autohaendler (Gebrauchtwagen, Neuwagen)
Gebrauchtwagen -> gebrauchtwagen(Gebraucht)
Neuwagen       -> neuwagen(Neu)
Gebraucht      -> fabrikat(Modell, Baujahr)
Neu            -> fabrikat(Modell, Rabatt)
Modell         -> modell()
Alter          -> baujahr()
Rabatt         -> rabatt()
```

Wo liegen die Schwächen von Baumgrammatiken mit eindeutigen Typen?

Beispiel 4.6 Wir betrachten die folgende Sprache L_2, die aus Bäumen der Tiefe zwei besteht: Alle Bäume besitzen die Wurzel `neuwagen`, die zwei Kinder mit dem identischen Namen `fabrikat` besitzt. Ein `fabrikat`-Knoten besitzt entweder ein Kind mit Namen `modell` oder zwei Kinder mit den Namen `modell` und `rabatt`. Zusätzlich fordern wir, dass jeder Baum in L_2 mindestens ein „`rabatt`-Blatt" besitzt.



Wir zeigen, dass L_2 eine reguläre Baumsprache ist und geben dazu die folgenden Produktionen an.

```
Neuwagen       -> neuwagen((Neu,Neur) | (Neur,Neu) | (Neur,Neur) )
Neu            -> fabrikat(Modell)
Neur           -> fabrikat(Modell,Rabatt)
Modell         -> modell()
Rabatt         -> rabatt()
```

Diese spezielle Baumgramatik ist natürlich nicht eindeutig, da die im Inhalt von `Neuwagen` auftretenden Variablen `Neu` und `Neur` im „Buchstaben" `fabrikat` kollideren.

Aufgabe 64
Zeige, dass L_2 keine Baumsprache mit eindeutigen Typen ist.

Es ist also nicht möglich, eine Klasse von XML Schema Dokumenten zu definieren, das alle Autohändler aufführt, die mindestens einen Neuwagen mit Rabatt anbieten. Für reguläre Baumgrammatiken hingegen stellt dies kein Problem dar: Sie können die globale Eigenschaft „mindestens einmal eine Eigenschaft zu erfüllen" beschreiben:

Aufgabe 65
Die Sprache L bestehe aus allen Bäumen, deren Knoten mit entweder `a` oder `b` beschriftet sind. Zusätzlich verlangen wir, dass mindestens ein Knoten mit `a` markiert ist.

Zeige, dass L eine reguläre Baumsprache ist.

Lemma 4.7

(a) Die Klasse der lokalen Baumsprachen ist eine echte Teilmenge der Klasse der Baumsprachen mit eindeutigen Typen.

(b) Lokale Baumsprachen und Baumsprachen mit eindeutigen Typen sind abgeschlossen unter Durchschnitt, nicht aber unter Vereinigung und Komplementen.

Beweis (a) Jede lokale Baumgrammatik ist auch eine Baumgrammatik mit eindeutigen Typen. Beispiel 4.5 zeigt, dass es Baumsprachen mit eindeutigem Typ gibt, die keine lokalen Baumsprachen sind.

(b) Der Nicht-Abschluss unter Vereinigung wird in der folgenden Aufgabe nachgewiesen. Für die verbleibenden Aussagen verweisen wir auf die Literatur. □

Aufgabe 66
Die Sprachen $L_1 = \{B_1\}$ und $L_2 = \{B_2\}$ bestehen jeweils aus genau einem Baum der Tiefe zwei; alle Knoten sind mit dem Buchstaben a beschriftet. B_1 ist der vollständige binäre Baum. B_2 hat ebenfalls eine Wurzel mit zwei Kindern, die beiden Kinder haben aber selbst nur jeweils ein Kind.

Zeige: L_1, L_2 sind lokale Baumsprachen, $L_1 \cup L_2$ ist keine Baumsprache mit eindeutigem Typ.

Also sind weder lokale Baumsprachen noch Baumsprachen mit eindeutigem Typ unter Vereinigung abgeschlossen.

Trotz gewonnener Ausdrucksstärke erlauben Baumsprachen mit eindeutigen Typen weiterhin eine pfeilschnelle Lösung des Typechecking Problems.

Ein Parser für XML Schema, bzw. für Baumsprachen mit eindeutigen Typen

Das XML-Dokument D liege mit einer XML Schema Deklaration, also einer eindeutigen Baumgrammatik G vor. Typechecking wird wieder durch einzigen links-nach-rechts Durchlauf von D durchgeführt:

(1a) Wenn die Wurzelkennung mit Namen a angetroffen wird, dann suche eine S-Produktion $S \to aR$. Wenn es keine solche Produktion gibt, dann brich mit einer Fehlermeldung ab.

(1b) Wenn eine von der Wurzelkennung verschiedene Startkennung mit Namen a angetroffen wird, dann haben wir bereits die Startkennung des Elternknotens gelesen und genau eine Produktion $Y \to bR'$ für den Elternknoten ausgewählt. Wir wählen eine Produktion $X \to aR$ aus, so dass X in R' auftaucht. Wenn es keine solche Produktion gibt, dann brich mit einer Fehlermeldung ab.

/* Da die Baumgrammatik eindeutig ist, gibt es keine zwei Variablen, die in R' auftauchen und im Buchstaben a kollidieren. Unsere Wahl ist also eindeutig! */

(2) Wenn die entsprechende Endkennung gelesen wird, dann überprüfe, ob die den Kindern zugeordnete Buchstabenfolge den regulären Ausdruck R erfüllt. Ist dies nicht der Fall, dann brich mit einer Fehlermeldung ab.

4.3.2 Reguläre Baumgrammatiken und Baumautomaten

Wir betrachten jetzt uneingeschränkte reguläre Baumsprachen, die die Ausdruckskraft von Schema Sprachen wie RELAX NG[3] charakterisieren. Wir beschränken uns auf beschrifte Binärbäume.

Ein Parser für reguläre Baumsprachen

Das XML-Dokument D werde durch die reguläre Baumgrammatik G erzeugt. Auch diesmal genügt ein links-nach-rechts Durchlauf von D:

(1) Wenn eine Startkennung mit Namen a angetroffen wird, dann betrachte alle Produktionen $X \to aR$.

/* Jetzt können wir uns nicht auf eine einzige Produktion konzentrieren. */

[3] Regular Language Description for XML New Generation

(2) Wenn die entsprechende Endkennung gelesen wird, dann wurden bereits die Kinder v_1, v_2 des aktuell betrachteten Knotens v verarbeitet. Wir nehmen an, dass wir für jedes Kind v_i eine Menge Kandidaten(v_i) von Variablen bestimmt haben.

/* In Kandidaten(v_i) sammeln wir alle Variablen, die für die Erzeugung von v_i in Betracht kommen. */

Wir nehmen eine Variable X in die Menge Kandidaten(v) auf, wenn es eine Produktion $X \to aR$ und Variablen $Y \in$ Kandidaten(v_1), $Z \in$ Kandidaten(v_2) gibt, so dass YZ den regulären Ausdruck R erfüllt.

Gib eine Fehlermeldung aus, wenn Kandidaten(v) leer ist.

(3) Typechecking ist erfolgreich, wenn das Startsymbol in der Kandidatenmenge der Wurzel vorkommt.

Für jeden Knoten v müssen wir bis zu $|P| \cdot |V|^2$ Mal überprüfen, ob ein regulärer Ausdruck erfüllt wird. Typechecking kann also weiterhin effizient durchgeführt werden, allerdings wächst der Aufwand beträchtlich. Beachte, dass wir zum ersten Mal möglicherweise mehrere Ableitungen erhalten: Unterschiedliche Interpretationen desselben Dokuments können für nachfolgende XML-Anwendungen problematisch sein und sollten zu einer Warnung des XML-Parsers führen. Diese mögliche Mehrdeutigkeit wie auch das verlangsamte Typechecking sind die Schwachstellen allgemeiner regulärer Baumgrammatiken, ihre Stärken liegen in ihrer Ausdruckskraft.

Auch allgemeine reguläre Baumsprachen, also reguläre Baumsprachen mit möglicherweise nichtbinären Bäumen besitzen effiziente Parser. Die eher ärgerlichen technischen Details besprechen wir hier nicht.

Wie groß ist die Klasse der regulären Baumsprachen und welche Eigenschaften hat die Klasse? Um diese Frage zu beantworten, setzen wir Baumautomaten ein. Angenommen, ein beschrifteter binärer Baum B wird von einer regulären Baumgrammatik $G = (V, \Sigma, S, P)$ erzeugt. Wir stellen uns vor, dass für jeden Buchstaben $a \in \Sigma$ ein Teilmenge $Q(a)$ aus einer Zustandsmenge ausgewählt wird. Für jedes Blatt von B ersetzen wir die Beschriftung a des Blatts durch einen (nichtdeterministisch geratenen) Zustand $q(a) \in Q(a)$. Ein Baumautomat berechnet jetzt (ebenfalls nichtdeterministisch) sukzessive Zustände für alle inneren Knoten v. Wenn v die Kinder v_1, v_2 besitzt und

- wenn v_i den Zustand q_i erhalten hat
- und wenn v mit dem Buchstaben a beschriftet ist,

! dann erhält v nichtdeterministisch einen Zustand aus der Zustandsmenge $\delta(q_1, q_2, a)$.

Wir sagen, dass der Automat akzeptiert, wenn es eine Berechnung gibt, für die die Wurzel mit einem akzeptierenden Zustand beschriftet wird.

Definition 4.8 *Ein nichtdeterministischer Baumautomat* $A = (Q, \Sigma, \delta, (\, Q(a) \mid a \in \Sigma), F)$ *besitzt die folgenden Komponenten:*

- *Die endliche Zustandsmenge* Q *und das Alphabet* Σ,
- *das Programm* δ *mit* $\delta(q_1, q_2, a) \subseteq Q$ *für alle Zustände* $q_1, q_2 \in Q$ *und alle Buchstaben* $a \in \Sigma$,
- *für jeden Buchstaben* $a \in \Sigma$ *eine Menge* $Q(a) \subseteq Q$ *von Anfangszuständen und*
- *die Menge* $F \subseteq Q$ *akzeptierender Zustände.*

Der Automat ist genau dann deterministisch, wenn die Mengen $\delta(q_1, q_2, a)$ und $Q(a)$ für alle Zustände $q_1, q_2 \in Q$ und alle Buchstaben $a \in \Sigma$ einelementig sind.

$L(A)$ ist die Menge aller mit Buchstaben aus Σ beschrifteten Bäume, die von A akzeptiert werden.

Beispiel 4.7 Wir beschreiben einen deterministischen Baumautomaten A für die Sprache der binären Und/Oder Bäume aus Beispiel 4.4. A hat zwei Zustände `Null` und `Eins`, wobei `Null` Anfangszustand für den Buchstaben `0` und `Eins` Anfangszustand für den Buchstaben `1` ist. `Eins` ist der einzige akzeptierende Zustand. Hier ist das Programm:

$$\begin{aligned}
\delta(\texttt{Null}, \texttt{Null}, \texttt{or}) &= \{\texttt{Null}\},\\
\delta(\texttt{Eins}, \texttt{Null}, \texttt{and}) &= \delta(\texttt{Null}, \texttt{Eins}, \texttt{and}) = \delta(\texttt{Null}, \texttt{Null}, \texttt{and}) = \{\texttt{Null}\},\\
\delta(\texttt{Eins}, \texttt{Null}, \texttt{or}) &= \delta(\texttt{Null}, \texttt{Eins}, \texttt{or}) = \delta(\texttt{Null}, \texttt{Null}, \texttt{or}) = \{\texttt{Eins}\},\\
\delta(\texttt{Eins}, \texttt{Eins}, \texttt{and}) &= \{\texttt{Eins}\}.
\end{aligned}$$

Aufgabe 67

(a) Die Grammatik $G = (V, \Sigma, P, S)$ sei eine Grammatik mit eindeutigen Typen. Zeige, dass es einen deterministischen Baumautomaten A mit $|V|$ Zuständen gibt, so dass $L(G) = L(A)$ gilt.

(b) Die Baumsprache L werde von einem deterministischen Baumautomaten mit q Zuständen akzeptiert. Zeige, dass das Typechecking Problem für L und einen Baum B in Zeit $O(|B|)$ gelöst werden kann. Eine Vorverarbeitung, die nicht von B abhängt und in Zeit $O(q^2 \cdot |\Sigma|)$ abläuft, ist erlaubt.

Gibt es zu jeder regulären Baumgrammatik einen äquivalenten Baumautomaten und umgekehrt, gibt es zu jedem Baumautomaten eine äquivalente reguläre Baumgrammatik? Ja! Wenn zum Beispiel eine reguläre Baumgrammatik $G = (V, \Sigma, P, S)$ gegeben ist, dann entwerfen wir einen Baumautomaten mit Zustandsmenge $Q = V$ und dem Programm δ, so dass

$$\begin{aligned}
\delta(Y, Z, a) = \{\, X \in V \quad | \quad & X \to aR \text{ ist eine Produktion in } P \text{ und}\\
& YZ \text{ erfüllt den regulären Ausdruck } R\,\}.
\end{aligned}$$

Welche Anfangszustände sollten wir für den Buchstaben a wählen? Wir setzen

$$Q(a) = \{X \in V \mid X \to a \text{ ist eine Produktion in } P\,\}.$$

Der Automat rät also in jedem Fall eine Variable, die die aktuelle Beschriftung verursacht haben könnte. Sind alle Vermutungen des Automaten richtig, dann muss der Baum aus der Grammatik ableitbar sein, solange der Automat das Startsymbol S für die Wurzel geraten hat. Wir setzen deshalb

$$F = \{S\}.$$

Wenn aber der Baum aus den Regeln der Grammatik ableitbar ist, dann muss es auch eine Folge richtiger Vermuungen geben.

Wir haben gezeigt, dass es zu jeder regulären Baumgrammatik G einen nichtdeterministischen Baumautomaten A mit $L(G) = L(A)$ gibt. Auch die Umkehrung ist richtig:

Aufgabe 68

Sei A ein nichtdeterministischer Baumautomat. Zeige: Dann gibt es eine reguläre Baumgrammatik G mit $L(A) = L(G)$.

Satz 4.9 *Baumautomaten akzeptieren genau die Klasse der regulären Baumsprachen.*

Baumgrammatiken erzeugen Bäume B nach dem top-down Muster, denn die Erzeugung von B beginnt mit der Wurzel und endet an den Blättern. Baumautomaten hingegen arbeiten nach dem bottom-up Prinzip, denn Baumautomat durchforsten B beginnend mit den Blättern und arbeiten sich zur Wurzel durch. Satz 4.9 zeigt also, dass die top-down mit der bottom-up Sichtweise zusammenfällt.

Wir können natürlich auch „top-down Baumautomaten“ A definieren:

> Wenn A die Zustandsmenge Q besitzt, dann weist das Programm δ jedem Paar $(q, a) \in Q \times \Sigma$ eine Teilmenge $\delta(q, a) \subseteq Q \times Q$ zu.
>
> Wie arbeitet A auf einem Binärbaum B? A beginnt an der Wurzel und arbeitet sich nichtdeterministisch bis zu den Blättern durch. Wenn A den Knoten v mit Namen a erreicht hat und sich im Zustand q befindet, dann wählt A nichtdeterministisch ein Paar $(q_1, q_2) \in \delta(q, a)$ aus und weist q_1 dem linken Kind und q_2 dem rechten Kind zu. B wird akzeptiert, wenn alle den Blättern zugewiesenen Zustände akzptierend sind.
>
> Schließlich definiere $L(A)$ als die Menge aller von A akzeptierten Binärbäume.

Aufgabe 69
Sei A ein (bottom-up) Baumautomat. Zeige, dass es einen top-down Baumautomaten A' mit $L(A) = L(A')$ gibt.

Aufgabe 70
Sei A ein top-down Baumautomat. Zeige, dass es einen (bottom-up) Baumautomaten A' mit $L(A) = L(A')$ gibt.

Aufgabe 71
Die reguläre Baumsprache L bestehe aus zwei Bäumen B_1 und B_2. Der Baum B_1 besteht aus der Wurzel und den mit a und b beschrifteten Blättern. Baum B_2 stimmt mit B_1 überein, allerdings ist die Reihenfolge der Blattmarkierungen vertauscht. Zeige, dass es keinen deterministischen top-down Automaten gibt, der L akzeptiert.

Nichtdeterministische top-down Automaten erzeugen also genau die Klasse der regulären Baumsprachen. Wenn wir allerdings mit deterministischen top-down Baumautomaten arbeiten, dann verlieren wir Ausdruckskraft!

Für konventionelle Baumautomaten, also für bottom-up Baumautomaten, ist die Situation gänzlich anders: Zu jedem nichtdeterministischen Baumautomaten N gibt es einen äquivalenten deterministischen Baumautomaten D. Wir gehen genauso vor wie im Fall von Worten: Insbesondere wird D für jeden Knoten eines Baums die *Menge* der von N zugewiesenen Zustände als Zustand annehmen, wir verwenden also wieder die Potenzmengenkonstruktion.

Satz 4.10 *Zu jedem nichtdetermministischen Baumautomaten N gibt es einen äquivalenten deterministischen Baumautomaten D.*

Jetzt können wir die Ernte einfahren und erhalten die folgenden Abschlusseigenschaften.

Satz 4.11 *Wenn L_1 und L_2 reguläre Baumsprachen sind, dann sind auch $L_1 \cup L_2, L_1 \cap L_2$ und $\overline{L_1}$ reguläre Baumsprachen.*

Beweis Angenommen, der nichtdeterministische Baumautomat N_i akzeptiert L_i. Dann bauen wir einen nichtdeterministischen Baumautomaten N für $L_1 \cup L_2$ wie folgt: N entscheidet sich nichtdeterministisch zu Anfang, ob er N_1 oder N_2 simulieren soll. Danach führt er die Simulation durch und akzeptiert, wenn der simulierte Austomat akzeptiert.

Um $\overline{L_1}$ zu akzeptieren, wählen wir einen deterministischen Baumautomaten D für L_1 und akzeptieren genau dann, wenn D verwirft.

Schließlich beachte $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ und auch der Abschluss unter Durchschnitten folgt. □

Arbeitet man mit regulären Baumgrammatiken, dann lassen sich also problemlos zwei Dokumenttypen zu einem neuen Typ vereinigen.

Aufgabe 72
Zeige, dass das Leerheitsproblem:

Ist $L(N) \neq \emptyset$ für einen nichtdeterministischen Baumautomaten N?

effizient gelöst werden kann.

Aufgabe 73
Für einen beschrifteten Baum B ist string(B) die Konkatenation aller Blattbeschriftungen, wobei die Reihenfolge der Blätter durch einen Präorder-Durchlauf (gemäß der Ordnung des Baums) definiert ist.

(a) Zeige, dass

$$\text{string}(L) = \{\, \text{string}(B) \mid B \in L \,\}$$

kontextfrei ist, wenn L eine reguläre Baumsprache ist.

(a) Zeige, dass es für jede kontextfreie Sprache K eine reguläre Baumsprache L mit

$$K = \text{string}(L)$$

gibt.

4.4 XPath und XQuery

Wir geben eine kurze Beschreibungen der Sprachen XPath und XQuery, die für die Navigation, bzw. für Datenbankanfragen in XML-Dokumenten eingesetzt werden. Beide Sprachen verwenden reguläre Ausdrücke als ihre zentralen Beschreibungsmechanismen.

XPath: Navigation in XML-Dokumenten

Die XPath-Ausdrücke sind ähnlich aufgebaut wie die Beschreibung von Verzeichniswegen in Unix. Exemplarisch nehmen wir an, dass die Elemente `a,b,c,d,e,f,g` in einem vorgegebenen XML-Dokument vorkommen und dass `x` ein Attributwert ist. Wir betrachten einen „typischen“ XPath Ausdruck

```
//a/b[c/d]//e[f=x]/g
```

`//` ist ein Nachfahren-Operator und `/` ist ein Kindoperator; die eckigen Klammern beschreiben jeweils einen Filter. Da der Ausdruck mit einem Operator beginnt, nimmt XPath an, dass die Navigation in der Wurzel beginnt und bestimmt *alle* Knoten, die Endpunkte der vom Ausdruck beschriebenen Wege sind. Welche Wege beschreibt der Ausdruck?

(1) Wir beachten zuerst die Filter nicht.

- Jeder Weg beginnt mit einem Nachfahren der Wurzel, der mit Element `a` beschriftet ist und ein mit `b` beschriftetes Kind u besitzt.
- u muss einen Nachfahren v haben, der mit Element `e` beschriftet ist.
- Schließlich muss v ein mit `g` beschriftetes Kind w besitzen.

(2) Die Filter kommen jetzt ins Spiel. Nur die Wege überleben

- für die der Knoten u auch ein mit `c` beschriftetes Kind hat, das seinerseits ein mit `d` beschriftetes Kind besitzt.
- Zuletzt wird gefordert, dass v ein mit `f` beschriftetes Kind besitzt und dass dieses Kind den Attributwert `x` hat.

Die Ausgabe für den Ausdruck besteht dann aus allen Endpunkten w der überlebenden Wege.

Nicht erwähnt haben wir die Möglichkeit, Attribute mit dem `@`-Operator auszuwählen. Wenn wir zum Beispiel die Namen aller Zutaten für das Pfannkuchenrezept ausgeben wollen, dann gelingt dies mit dem Ausdruck `//zutat/@name`. Hätten wir hingegen eine Wurzel `rezepte` mit Kindern vom „Typ" `rezept` definiert, so müsste man diesmal den Ausdruck `/rezept[@name=Mein erster Pfannkuchen]//zutat/@name` verwenden.

Viele Anfragen von XPath können auch durch Baumautomaten implementiert werden. Für einen nichtdeterministischen Baumautomaten A zeichnen wir dazu eine Teilmenge $S \subseteq Q$ von „auswählenden Zuständen" aus. Wir sagen dann, dass A einen Knoten v auswählt, wenn es eine akzeptierende Berechnung von A gibt, die v mit einem Zustand aus S durchläuft.

XQuery: Eine Anfragesprache für XML-Datenbanken

XQuery ist eine „vollwertige" Anfragesprache, das Analogon von SQL für relationale Datenbanken. Xquery benutzt XPath als „Navigationskomponente" und verwendet unter anderem for-let-where-return Ausdrücke (oder FLWR-Ausdrücke):

```
for $variable1 in AUSDRUCK1
let $variable2 := AUSDRUCK2
where BOOLESCHER AUSDRUCK
return AUSDRUCK3
```

Hier darf jeder XQuery Ausdruck benutzt werden: Da ein XPath Ausdruck auch ein XQuery Ausdruck ist, dürfen somit zum Beispiel alle XPath Ausdrücke benutzt werden. Auch kann zum Beispiel ein FLWR-Ausdruck als Ausdruck in einem anderen FLWR-Ausdruck vorkommen.

Wir geben ein Beispiel für die Datenbank einer Bibiothek, in der jedes Buch mit Autor, Titel und Verlag aufgeführt ist. Wir möchten die Verlage herausfinden, die mindestens 100 in der Bibiothek geführte Bücher veröffentlicht haben. Wir benutzen dazu die Funktionen `distinct-values` und `count`:

```
for $v in distinct-values(document("bib.xml")//Verlag)
let $b := document("bib.xml")//book[publisher = $v]
where count($b) > 100
return $v
```

In freier Übersetzung: Für jeden Verlag, der in der Datenbank `bib.xml` vertreten ist, betrachten wir alle von diesem Verlag publizierten Bücher. Das `bib.xml`-Dokument, eingeschränkt auf alle Verlage, die mindestens 100 Bücher der Datenbank veröffentlicht haben, wird ausgegeben.

Während also XPath für ein vorgegebenes XML-Dokument eine Teilmenge von Knoten ausgibt, gibt XQuery ein modifiziertes XML-Dokument aus.

Teil II

Komplexitätsklassen

Kapitel 5

Speicherplatz-Komplexität

Das Ziel dieses Kapitels ist die Bestimmung der Speicherplatz-Komplexität, also die Bestimmung des für die Lösung eines algorithmischen Problems notwendigen und hinreichenden Speicherplatzes.

Warum sollte uns eine Untersuchung der Ressource „Speicherplatz“ interessieren? Es stellt sich heraus, dass die Speicherplatzkomplexität hilft, die Komplexität der Berechnung von Gewinnstrategien für viele nicht-triviale 2-Personen Spiele zu charakterisiere. Weitere algorithmische Probleme, deren Komplexität wir mit Hilfe der Speicherplatzkomplexität klären werden, sind:

(a) Akzeptiert ein nichtdeterministischer endlicher Automat eine gegebene Eingabe?

(b) Sind zwei nichtdeterministische endliche Automaten äquivalent?

(c) Minimiere einen NFA.

Des weiteren lassen sich „nicht-klassischen“ Berechnungsarten wie die Randomisierung oder Quantenberechnungen durch konventionelle deterministische Rechner simulieren, falls wir polynomiellen Speicherplatz erlauben. Im nächsten Kapitel werden wir zudem eine enge Verbindung zwischen der Speicherplatz-Komplexität eines Problems und seiner parallelen Rechenzeit feststellen.

Wir wählen deterministische I-O (input-output) Turingmaschinen als Rechnermodell. Eine I-O Turingmaschine M besitzt drei, nach links und rechts unbeschränkte ein-dimensionale Bänder mit jeweils einem Lese-Schreibkopf, wobei jeder Kopf in einem Schritt (höchstens) zur jeweiligen linken oder rechten Nachbarzelle der gegenwärtig besuchten Zelle wandern darf.

(1) Das erste Band ist das Leseband. Wenn Σ_1 das Eingabealphabet bezeichnet, dann wird die Eingabe $w = (w_1, \ldots, w_n) \in \Sigma_1^*$ auf den Zellen $0, \ldots, n+1$ wie folgt abgespeichert: Die Zellen 0 und $n+1$ enthalten das Begrenzersymbol #, während Zelle i $(1 \leq i \leq n)$ mit dem Buchstaben w_i beschriftet ist. Der Kopf des Lesebands befindet sich zu Anfang der Berechnung auf Zelle 0 und darf während der Berechnung weder den durch die Begrenzer beschriebenen Bereich verlassen noch einen Zelleninhalt überschreiben. Der Kopf verhält sich somit wie ein Lesekopf.

(2) Das zweite Band ist das Arbeitsband, dessen Zellen gelesen aber auch mit den Buchstaben des Arbeitsalphabets Γ, beschrieben werden dürfen. Zu Anfang der Berechnung sind alle Zellen mit dem Blanksymbol `B` beschriftet.

(3) Das dritte Band ist das Ausgabeband, dessen Zellen zu Anfang der Berechnung ebenfalls mit dem Blanksymbol beschriftet sind. Der Kopf darf das Ausgabeband mit Buchstaben eines Ausgabealphabets Σ_2 beschreiben, wobei aber in einem Schritt nur die rechte Nachbarzelle aufgesucht werden darf. Der Kopf verhält sich also wie ein links-nach-rechts wandernder Schreibkopf.

Wenn die Maschine M auf Eingabe w hält, dann ist die Konkatenation der Inhalte der Zellen $0, \ldots m$ die Ausgabe $M(w)$ der Berechnung. (Die Zelle 0 ist die zu Anfang besuchte Zelle und die Zelle m ist die im letzten Schritt besuchte Zelle des Ausgabebands.)

Wenn die Maschine M für jede Eingabe nur die erste Zelle des Ausgabebands modifiziert und dabei nur die Symbole 0 oder 1 druckt, dann sagen wir, dass die Eingabe w akzeptiert (Ausgabe 1) bzw. verworfen wird (Ausgabe 0). Wir definieren

$$L(M) = \{w \in \Sigma_1^* \mid \ M \text{ akzeptiert } w \ \}$$

als die von M erkannte Sprache.

I-O Turingmaschinen erlauben die Definition des Speicherplatzbedarfs durch Messung der Anzahl während der Berechnung besuchter Zellen des Arbeitsbands. Hätten wir nur 1-Band Turingmaschinen betrachtet, dann wäre der Speicherplatzbedarf stets mindestens so groß wie das Maximum von Ein- und Ausgabelänge, und wir hätten den teuren Arbeitsspeicher mit billigen Hintergrundspeichern (für die Ein- und Ausgabe) gleichgesetzt.

Definition 5.1 *Sei Σ ein Alphabet.*

(a) Die I-O Turingmaschine M besuche für jede Eingabe $w \in \Sigma^*$ genau $\texttt{Dspace}_M(w)$ viele verschiedene Zellen des Arbeitsbands. Wir definieren

$$\texttt{Dspace}_M(n) = \max\{\texttt{Dspace}_M(w) \mid w \in \Sigma^n\}$$

als den Speicherplatzbedarf von M für Eingabelänge n.

(b) Sei $s : \mathbb{N} \to \mathbb{N}$ gegeben. Dann definieren wir die Klasse aller mit Platzverbrauch höchstens $O(s(n))$ lösbaren Entscheidungsprobleme durch

$$\texttt{Dspace}(s) = \{L \subseteq \Sigma^* \mid L(M) = L \textit{ für eine I-O TM } M \textit{ mit } \texttt{Dspace}_M = O(s) \ \}.$$

(c) Die Komplexitätsklasse DL *besteht aus allen mit logarithmischem Speicherplatzbedarf erkennbaren Sprachen, also*

$$\texttt{DL} = \texttt{Dspace}(\log_2 n).$$

(d) Die Komplexitätsklasse PSPACE *besteht aus allen mit polynomiellem Speicherplatzbedarf erkennbaren Sprachen, also*

$$\texttt{PSPACE} = \bigcup_{k \in \mathbb{N}} \texttt{Dspace}(n^k).$$

Man beachte, dass wir die Speicherplatzkomplexität in Teil (b) asymptotisch definiert haben. Dieses Vorgehen wird durch das folgende "Kompressionslemma" geradezu erzwungen.

Lemma 5.2 *Die I-O Turingmaschine M erkenne die Sprache L mit Speicherplatzbedarf höchstens $s(n)$. Dann gibt es eine I-O Turingmaschine M', die L mit Speicherplatzbedarf höchstens $\lceil \frac{s(n)}{2} \rceil$ erkennt.*

Beweisskizze: Sei Γ das Arbeitsalphabet von M. Die Maschine M' verhält sich genau wie M, benutzt aber das Arbeitsalphabet Γ^2. $\square$

Wie ist der Speicherplatzbedarf von nichtdeterministischen Turingmaschinen zu definieren? Offensichtlich ist auch hier die Betrachtung von (nichtdeterministischen) I-O Turingmaschinen M angesagt. Wir definieren den Speicherplatzbedarf $\texttt{Nspace}_M(w)$ für eine Eingabe w durch den maximalen Speicherplatzbedarf einer Berechnung für w.

Definition 5.3 *Sei Σ ein Alphabet.*

(a) *Wir definieren*

$$\texttt{Nspace}_M(n) = \max\{\texttt{Nspace}_M(w) \mid w \in \Sigma^n\}$$

als den Speicherplatzbedarf von M für Eingabelänge n.

(b) *Sei $s : \mathbb{N} \to \mathbb{N}$ gegeben. Dann definieren wir*

$$\texttt{Nspace}(s) = \{L \subseteq \Sigma^* \mid \text{es gibt eine nichtdeterministische I-O TM } M \text{ mit } L(M) = L \text{ und } \texttt{Nspace}_M = O(s)\ \}.$$

(c) *Die Komplexitätsklasse* NL *besteht aus allen nichtdeterministisch, mit logarithmischem Speicherplatzbedarf erkennbaren Sprachen, also*

$$\texttt{NL} = \texttt{Nspace}(\log_2 n).$$

(d) *Die Komplexitätsklasse* NPSPACE *besteht aus allen nichtdeterministisch, mit polynomiellem Speicherplatzbedarf erkennbaren Sprachen, also*

$$\texttt{NPSPACE} = \bigcup_{k \in \mathbb{N}} \texttt{Nspace}(n^k).$$

5.1 Eine Platzhierarchie

Definition 5.4 *Eine Funktion $s : \mathbb{N} \to \mathbb{N}$ heißt genau dann platz-konstruierbar, wenn es eine deterministische Turingmaschine gibt, die für eine jede Eingabe der Länge n höchstens $O(s(n))$ Speicherplatz benötigt, um $s(n)$ Zellen zu markieren.*

Wir geben mit Hilfe der Diagonalisierungsmethode eine Speicherplatz-Hierarchie an und zeigen, dass man mit mehr zur Verfügung stehendem Speicher mehr Entscheidungsprobleme lösen kann.

Satz 5.5 *(Das Speicherplatz-Hierarchie Theorem)*

Die Funktion s sei platz-konstruierbar. Dann ist $\texttt{Dspace}(o(s))$ *eine echte Teilmenge von* $\texttt{Dspace}(s)$.

Beweis: Für eine platz-konstruierbare Funktion s konstruieren wir die folgende I-O Turingmaschine M^*.

Algorithmus 5.1 *Die Diagonalisierungsmethode*

(1) M^* bestimmt die Länge n der Eingabe w.

(2) *M^* steckt auf dem Arbeitsband $2s(n)$ Zellen ab.*

/ Dies ist mit Speicherplatz $O(s(n))$ möglich, da s platz-konstruierbar ist. */*

(3) *Wenn w nicht von der Form $\langle M \rangle 0^k$ für eine I-O Turingmaschine M und eine Zahl $k \in \mathbb{N}$ ist, dann verwirft M^*.*

/ $\langle M \rangle$ bezeichnet die Gödelnummer der Turingmaschine M. */*

(4) *M^* simuliert M auf Eingabe $w = \langle M \rangle 0^k$ und beginnt die Simulation mit dem Kopf in der Mitte des abgesteckten Bereichs. M^* verwirft, wenn M irgendwann den abgesteckten Bereich verlässt oder mehr als $2^{s(n)}$ Schritte benötigt.*

/ Kann M^* auf Speicherplatz $s(n)$ gleichzeitig einen bis $2^{s(n)} - 1$ zählenden Zähler implementieren und eine $s(n)$ platzbeschränkte Berechnung von M simulieren? Ja, das ist bei einem entsprechend vergrößerten Arbeitsalphabet unproblematisch. */*

(5) *M^* akzeptiert w, wenn M verwirft. Ansonsten akzeptiert M und M^* wird verwerfen.*

Wir beachten zuerst, dass M^* immer hält und mit Speicherplatzbedarf $O(s(n))$ auskommt. Also ist $L(M^*) \in \texttt{Dspace}(s)$.

Warum liegt $L(M^*)$ nicht in $\texttt{Dspace}(r)$ für eine Funktion $r = o(s)$? Ist dies der Fall, dann wird $L(M^*)$ von einer I-O Turingmaschine M mit Speicherplatzbedarf r erkannt. Für hinreichend große Eingabelänge n rechnet M stets in Zeit höchstens 2^s und M^* simuliert M für Eingaben $w = \langle M^* \rangle 0^k$ mit hinreichend großem k erfolgreich. Jetzt garantiert Schritt (5), dass sich $L(M)$ und $L(M^*)$ unterscheiden: Speicherplatz $r = o(s)$ ist unzureichend für die Berechnung von $L(M^*)$. □

Aufgabe 74
Zeige das Platz-Hierarchie Theorem für nichtdeterministische Turingmaschinen: $\texttt{Nspace}(s)$ ist eine echte Teilmenge von $\texttt{Nspace}(S)$, falls $s = o(S)$ und falls S platz-konstruierbar ist.

5.2 Sub-Logarithmischer Speicherplatz

Was ist die Speicherplatzkomplexität regulärer Sprachen? Eine reguläre Sprache L werde durch einen endlichen Automaten A akzeptiert. Wir fassen A als eine I-O Turingmaschine auf, die wie A programmiert wird. Zusätzlich erzwingen wir, dass das Symbol 1 (bzw) 0) auf das Ausgabeband gedruckt wird, wenn sich A vor dem Lesen des rechten Begrenzungssymbol in einem akzeptierenden (bzw. verwerfenden) Zustand befindet. Die I-O Turingmaschine benötigt das Arbeitsband nicht und $L \in \texttt{Dspace}(0)$ folgt.

Überraschenderweise führt sehr(!) geringer Speicherplatz im Vergleich zu leerem Speicher nicht zu größerer Berechnungskraft wie das nächste Ergebnis zeigt.

Satz 5.6 *(Sprachen mit sehr geringem Speicherplatzbedarf sind regulär)*
Die I-O Turingmaschine M akzeptiere L mit Speicherplatzbedarf $s_M = o(\log_2 \log_2 n)$. Dann ist L regulär. Insbesondere folgt für jede Funktion $s : \mathbb{N} \to \mathbb{N}$ mit $s = o(\log_2 \log_2 n)$, dass

$$\texttt{Dspace}(s) = \texttt{Dspace}(0) = \textit{ die Klasse der regulären Sprachen.}$$

Beweisskizze: Wir beschränken uns auf die Untersuchung von I-O Turingmaschine M mit konstantem Speicherplatzbedarf. Da der Speicherplatzbedarf nicht von der Eingabelänge abhängt,

können wir die möglichen Speicherinhalte als Zustände in die Programmierung einer äquivalenten I-O Turingmaschine M' aufnehmen, wobei M' keinerlei Speicherbedarf hat. Damit ist M' aber nichts anderes als ein deterministischer Zwei-Weg Automat, also „ein endlicher Automat, der die Eingabe in jeder Richtung lesen darf".

Ein Zwei-Weg Automat Z lässt sich aber durch einen NFA N mit ε-Übergängen simulieren. Dazu beachten wir zuerst, dass sich der Automat Z in einer Endlos-Schleife befindet (und damit nicht hält), wenn derselbe Zustand zweimal über derselben Eingabeposition angenommen wird. Q sei die Zustandsmenge von Z und es gelte $|Q| = q$. Wir wählen

$$Q' = \bigcup_{k=1}^{q} \Sigma \times (Q \times \{\text{links, rechts}\})^k$$

als Zustandsmenge von N. Wenn N den Zustand $(a, q_1, \text{richtung}_1, \ldots, q_r, \text{richtung}_r)$ annimmt, dann spekuliert N, dass

(a) dass a der Inhalt der gegenwärtig besuchten Zelle ist,

(b) Z den Zustand q_i beim $i$ten Besuch der Zelle annimmt,

(c) und dass der $i$te Besuch von der linken Nachbarzelle ($\text{richtung}_i = \text{links}$), bzw. der rechten Nachbarzelle ($\text{richtung}_i = \text{rechts}$) erfolgt.

Wir erlauben einen Zustandsübergang

$$(a, q_1, r_1, \ldots, q_s, r_s) \xrightarrow{N} (b, q_1', r_1', \ldots, q_t', r_t')$$

bei gelesenem Buchstaben a, wenn

(a) N die Vektoren $\vec{q}$ und $\vec{r}$ unter der Annahme verifizieren kann, dass die Vektoren $\vec{q'}$ und $\vec{r'}$ richtig geraten wurden und wenn

(b) für alle i mit „$r_i' = \text{links}$" der Zustand q_i' richtig geraten wurde.

N wird nur dann akzeptieren, wenn „$r_i = \text{links}$" für alle i und wenn der finale Zustand akzeptierend ist.

M' (und damit auch M) akzeptiert somit eine reguläre Sprache. $\square$

Beispiel 5.1 $\texttt{Dspace}(\log_2 \log_2 n)$ enthält auch nicht-reguläre Sprachen. Es sei $\text{bin}(i)$ die Binärdarstellung der Zahl i ohne führende Nullen. Wir wählen $\Sigma = \{0, 1, \$\}$ als Eingabealphabet und definieren die Sprache

$$\text{BIN} = \{\text{bin}(1)\$\text{bin}(2)\$ \cdots \$\text{bin}(n) \mid n \in \mathbb{N}\}.$$

Man überzeugt sich leicht mit dem Pumping Lemma, dass BIN nicht regulär ist. Es bleibt zu zeigen, dass $\text{BIN} \in \texttt{Dspace}(\log_2 \log_2 n)$. Das ist aber klar(?), da $\text{bin}(i)$ für jedes $i \leq n$ aus höchstens $\lceil \log_2 n \rceil$ Bits besteht.

Insbesondere haben wir für die Sprache BIN das seltene Glück, die genaue Speicherkomplexität angeben zu können. Denn da BIN nicht regulär ist, folgt $\text{BIN} \notin \texttt{Dspace}(s)$, falls $s = o(\log_2 n \log_2 n)$. Andererseits haben wir gerade $\text{BIN} \in \texttt{Dspace}(\log_2 \log_2 n)$ nachgewiesen.

Die sublogarithmischen Speicher-Komplexitätsklassen sind aber recht mickrig und erst $\texttt{DL}$, also die logarithmische Speicherplatz-Komplexität, wird eine vernünftige Berechnungskraft besitzen.

5.3 Logarithmischer Speicherplatz

DL und NL gehören zu den wichtigsten Speicherplatz-Klassen.

- Die Berechnungskraft ist durchaus signifikant, da die Maschinen sich jetzt Positionen in der Eingabe merken können.
- Viele Eigenschaften, die für DL und NL gelten, verallgemeinern sich auf beliebige Speicherplatzklassen. Dieses Phänomen werden wir im Satz von Savitch und im Satz von Immerman-Szlepscenyi beobachten.

Wir beginnen mit deterministisch logarithmischem Platz.

5.3.1 DL

Sei PALINDROM die Sprache aller Palindrome über dem Alphabet $\{0,1\}$. Wir behaupten, dass sich PALINDROM mit logarithmischem Speicherplatz berechnen lässt.

Wir konstruieren eine I-O Turingmaschine, die zuerst die Eingabelänge n in Binärdarstellung abspeichert. Dies gelingt, indem ein Anfangs auf 0 gesetzter binärer Längen-Zähler sukzessive inkrementiert wird bis der Eingabekopf das Ende der Eingabe erreicht hat. Sodann werden nacheinander die Bitpositionen 1 und n, 2 und $n-1,\ldots,$ k und $n-k$ ... verglichen, indem der Eingabekopf jeweils um $n-1$ Positionen nach rechts, dann um $n-2$ Positionen nach links, um $n-3$ Positionen nach rechts, ... bewegt wird.

Zur Ausführung dieser Kopfbewegungen wird eine Kopie des Längen-Zählers angelegt und der Längen-Zähler wie auch die Kopie um Eins (auf $n-1$) vermindert. Die ersten Kopfbewegungen um $n-1$ Positionen nach rechts gelingen durch das Herunterzählen der Kopie. Nachdem das Ziel erreicht ist, wird der Längenzähler um Eins reduziert und der neue Inhalt in die Kopie geladen. Die nächsten $n-2$ Kopfbewegungen nach links, wie auch alle nachfolgenden Kopfbewegungen, werden dann mit demselben Verfahren gesteuert.

Die Klasse DL ist die erste nicht-triviale Speicherkomplexitätsklasse und enthält neben der Palindrom-Sprache weitere wichtige Sprachen wie die Dyck-Sprache aller wohlgeformten Klammerausdrücke oder die kontextsensitive Sprache $\{a^n b^n c^n \mid n \in \mathbb{N}\}$.

Aufgabe 75
$\{w \in \{(,)\}^* | w$ ist ein korrekter Klammerausdruck$\}$ ist die „Klammersprache“ . Dabei ist ein korrekter Klammerausdruck entweder

1. () oder
2. (A) für einen korrekten Klammerausdruck A oder
3. AB für korrekte Klammerausdrücke A, B.

Zeige, dass man die Klammersprache in DL entscheiden kann.

Wir können mit Hilfe des Konzepts der Konfiguration die Speicherplatzkomplexität der Palindrom-Sprache exakt bestimmen. (Zur Erinnerung: Die Konfiguration k_t zum Zeitpunkt t besteht aus der Position des Lesekopfs zum Zeitpunkt t, dem gegenwärtigen Zustand, der Position des Kopfs auf dem Arbeitsband und dem Inhalt des Arbeitsbands.) Es stellt sich heraus, dass logarithmische Speicherkomplexität nicht nur hinreichend, sondern auch notwendig ist. Dies bestätigt die Intuition, dass nicht-triviale Algorithmen zumindest die Fähigkeit haben sollten, sich an eine Eingabeposition erinnern zu können; beachte, dass diese Fähigkeit logarithmischen Speicherplatz voraussetzt.

Lemma 5.7 *Es ist PALINDROM* $\in$ DL. *Andererseits ist PALINDROM* $\notin$ Dspace(s), *falls* $s = o(\log_2 n)$.

Beweis: Sei M eine I-O Turingmaschine mit Speicherplatzbedarf $s(n)$, die die Palindrom-Sprache erkenne. M besitze q Zustände und ein Arbeitsalphabet der Größe γ. Die Eingabelänge sei gerade.

Behauptung 1: M wird die Eingabe-Position $\frac{n}{2}$ für mindestens eine Eingabe mindestens $\Omega(\frac{n}{s(n)})$ mal besuchen.

Bevor wir die Behauptung beweisen, zeigen wir, dass das Lemma aus der Behauptung folgt. Wir nehmen an, dass M den Speicherplatzbedarf $s(n) = o(\log_2 n)$ hat. Für jede Eingabe wird M dann, bei hinreichend großer Eingabelänge n, höchstens

$$q \cdot s(n) \cdot \gamma^{s(n)} = o(\sqrt{n})$$

Konfigurationen besitzen, für die der Lesekopf die Eingabeposition $\frac{n}{2}$ besucht. Aber Position $\frac{n}{2}$ kann nur einmal in einer vorgegebenen Konfiguration besucht werden, da ein zweiter Besuch in derselben Konfiguration zu einer Endlos-Schleife führt. Die Behauptung fordert aber eine Besuchshäufigkeit von mindestens $\Omega(\frac{n}{s(n)}) >> \sqrt{n}$ und wir haben einen Widerspruch zur Annahme $s(n) = o(\log_2 n)$ erhalten. □

Beweis von Behauptung 1: Wir weisen einer Eingabe w die Folge der Konfigurationen beim Besuch der Eingabe-Position $\frac{n}{2}$ zu. Wenn die Behauptung falsch ist, dann wird $\frac{n}{2}$ höchstens $o(\frac{n}{s(n)})$ mal besucht. Insgesamt gibt es aber mit (5.1) höchstens

$$(q \cdot \gamma^{s(n)} \cdot s(n))^{o(\frac{n}{s(n)})} = 2^{O(s(n)) \cdot o(\frac{n}{s(n)})} = 2^{o(n)}$$

Konfigurationenfolgen, wenn wir die Position des Lesekopfes fixieren. Es gibt aber $2^{n/2}$ Palindrome der Länge n und deshalb gibt es zwei verschiedene Palindrome u, v der Länge n mit identischer Konfigurationenfolge auf Position $\frac{n}{2}$.

Behauptung 2: Fooling Argument

Die Eingaben u, v mögen beide die Konfigurationenfolge k besitzen. Wenn $u = (u_1, u_2)$ und $v = (v_1, v_2)$ mit $|u_1| = |v_1| = n/2$, dann besitzt auch die Eingabe $w = (u_1, v_2)$ die Konfigurationenfolge k.

Aufgabe 76
Zeige Behauptung 2.

Behauptung 1 ist jetzt eine unmittelbare Konsequenz von Behauptung 2, da M für Palindrome u und v auch die Eingabe w akzeptiert, obwohl w kein Palindrom ist. □

Aufgabe 77
Die Palindrom-Sprache soll erkannt werden. Gib I-O-Maschinen an, die $PALINDROM$

(a) in Zeit $O(n)$ erkennen.

(b) in Zeit $O(n^2/\log n)$ mit Platz $O(\log n)$ erkennen.

Aufgabe 78
Zeige, dass $O(s)$ platzbeschränkten I-O-Turingmaschinen Zeit $\Omega(n^2/s)$ für das Erkennen der Palindrom-Sprache benötigen.

Hinweis: Betrachte Worte der Form $w\#^n w^R$ der Länge $3n$ (mit $w \in \{0,1\}^n$) und die Konfigurationenfolgen auf den Eingabepositionen $i \in \{n+1, \ldots, 2n\}$. Zeige, dass sich für alle Eingaben $w\#^n w^R$ und $v\#^n v^R$ (mit

$w \neq v$) auf Position i verschiedene Konfigurationsfolgen ergeben müssen. Daher muss es auf Position i mindestens 2^n verschiedene Konfigurationsfolgen geben und die meisten davon müssen lang sein. Schließe davon auf hohe Rechenzeit für die meisten Eingaben.

Wie mächtig ist DL?

Satz 5.8 *(Speicherplatz und Laufzeit)*

(a) Sei M eine I-O Turingmaschine, die mit Speicherplatzbedarf s arbeite. Dann ist die Laufzeit von M für Eingaben der Länge n durch $n \cdot 2^{O(s(n))}$ beschränkt.

(b) Es gelte $s(n) \geq \log_2 n$. Dann ist $\texttt{Dspace}(s) \subseteq \bigcup_{k \in \mathbb{N}} \texttt{Dtime}(2^{k \cdot s})$. Als Konsequenz folgt $\texttt{DL} \subseteq \texttt{P}$.

Beweis (a): Sei M eine I-O Turingmaschine mit Speicherplatzbedarf s und sei w eine beliebige Eingabe der Länge n. Wir beschreiben die Berechnung von M auf Eingabe w durch eine Folge von *Konfigurationen.* Die Konfiguration k_t zum Zeitpunkt t besteht aus der Position des Lesekopfs zum Zeitpunkt t, dem gegenwärtigen Zustand, der Position des Kopfs auf dem Arbeitsband und dem Inhalt des Arbeitsbands.

Wir kommen zur wesentlichen Beobachtung: M wird für keine Eingabe eine Konfiguration zweimal annehmen! Wäre dies nämlich der Fall, dann wird M in eine Endlos-Schleife gezwungen und hält nicht. Also ist die Laufzeit von M durch die Anzahl der Konfigurationen beschränkt. Wir nehmen an, dass M q Zustände und ein Arbeitsalphabet der Größe γ besitzt. Damit ist die Anzahl der Konfigurationen durch

$$n \cdot q \cdot s(n) \cdot \gamma^{s(n)} = n \cdot q \cdot s(n) \cdot 2^{s(n) \cdot \log_2 \gamma}$$

nach oben beschränkt und, da $s(n) \leq 2^{s(n)}$, erhalten wir

$$n \cdot q \cdot s(n) \cdot 2^{s(n) \cdot \log_2 \gamma} = n \cdot 2^{O(s(n))}. \tag{5.1}$$

Die Laufzeit ist natürlich durch die Anzahl der Konfigurationen beschränkt und die Behauptung folgt.

(b) ist eine direkte Konsequenz von Teil (a), denn wir nehmen $s(n) \geq \log_2 n$ an und $n \cdot 2^{O(s(n))} \leq 2^{O(s(n))}$ folgt. □

Wir definieren die Sprache U-REACHABILITY als die Menge aller ungerichteten Graphen G, die einen Weg von Knoten 1 nach Knoten 2 besitzen. Der Graph G werde durch seine Adjazenzmatrix repräsentiert. Analog sei die Sprache D-REACHABILITY definiert, wobei wir diesmal allerdings gerichtete Graphen betrachten.

Erst in 2004 konnte durch Omer Reingold gezeigt werden, dass U-REACHABILITY in DL liegt. Die Methode des Random Walks erlaubt eine mehr oder minder offensichtliche Lösung von U-REACHABILITY mit logarithmischem Speicherplatzbedarf, wenn wir zufällig arbeiten dürfen. Die von Reingold erhaltene Lösung für deterministische Turingmaschinen ist wesentlich komplizierter.

Offensichtlich ist D-REACHABILITY das schwierigere der beiden Probleme: Wir werden später starke Indizien erhalten, dass jede deterministische Turingmaschine mindestens die Speicherkomplexität $\Omega(\log_2^2 n)$ für die Lösung von D-REACHABILITY benötigt.

Wir erhalten aber eine Lösung von D-REACHABILITY mit logarithmischem Speicherplatzbedarf, wenn wir nichtdeterministische Turingmaschinen betrachten: Eine nichtdeterministische

I-O Turingmaschine rät einen Weg von Knoten 1 nach Knoten 2 und benutzt ihren logarithmischen Speicher zum Durchsuchen der Adjazenzmatrix. Dieser Algorithmus ist Anlaß, nichtdeterministische I-O Turingmaschinen zu betrachten.

Aufgabe 79
Gegeben sei ein gerichteter Graph als Adjazenzmatrix. Es soll entschieden werden, ob folgendes gilt: jeder Knoten hat höchstens einen Nachfolger und Knoten 2 kann von Knoten 1 erreicht werden.

Gib einen möglichst speichereffizienten deterministischen Algorithmus zur Lösung des Problems (auf I-O-Turingmaschinen) an.

Aufgabe 80
Im 2-Zusammenhangsproblem ist ein ungerichteter Graph als Adjazenzmatrix gegeben. Es soll entschieden werden, ob der Graph zweifach zusammenhängend ist, d.h., ob der Graph bei der Herausnahme eines beliebigen Knotens immer zusammenhängend bleibt.

Gib einen möglichst speichereffizienten deterministischen Algorithmus zur Lösung des Problems (auf I-O-Turingmaschinen) an.

5.3.2 NL und NL-Vollständigkeit

Ist D-REACHABILITY deterministisch mit logarithmischem Speicherplatz erkennbar? Wir werden im Folgenden zeigen, dass eine positive Antwort die Gleichheit der Klassen DL und NL erzwingt, und die wahrscheinliche Antwort ist also negativ. Insbesondere zeigen wir, dass D-REACHABILITY eine schwierigste Sprache in NL ist, wobei „Schwierigkeit“ durch LOGSPACE-Reduktionen gemessen wird.

Definition 5.9 *Seien Σ_1 und Σ_2 Alphabete und seien $L \subseteq \Sigma_1^*$, $K \subseteq \Sigma_2^*$ Sprachen über Σ_1 beziehungsweise Σ_2.*

(a) *Wir sagen, dass L LOGSPACE-reduzierbar auf K ist (geschrieben $L \leq_{\text{LOG}} K$), falls es eine (deterministische) I-O Turingmaschine M mit logarithmischem Speicherplatzbedarf gibt, so dass für alle Eingaben $w \in \Sigma_1^*$,*

$$w \in L \Leftrightarrow M(w) \in K.$$

(b) *Die Sprache K heißt NL-hart, falls $L \leq_{\text{LOG}} K$ für alle Sprachen $L \in$ NL gilt.*

(c) *Die Sprache K heißt genau dann NL-vollständig, wenn $K \in$ NL und wenn K NL-hart ist.*

Lemma 5.10 *(Die wesentlichen Eigenschaften der LOGSPACE-Reduktion)*

(a) Wenn $M \leq_{\text{LOG}} K$ und $K \leq_{\text{LOG}} L$, dann ist $M \leq_{\text{LOG}} L$.

(b) Wenn $L \leq_{\text{LOG}} K$ und wenn $K \in$ DL, dann ist auch $L \in$ DL.

Beweis: Übungsaufgabe. □

Wie auch für die Klassen P und NP (und die polynomielle Reduktion) fallen die Klassen DL und NL zusammen, wenn eine einzige NL-vollständige Sprache in DL liegt.

Korollar 5.11 *(LOGSPACE-vollständige und harte Sprachen)*

(a) Die Sprache K sei NL*-vollständig. Dann gilt*

$$K \in \mathtt{DL} \Leftrightarrow \mathtt{DL} = \mathtt{NL}.$$

(b) Wenn K NL*-hart ist und wenn $K \leq_{\text{LOG}} L$, dann ist auch L* NL*-hart.*

Gibt es überhaupt NL-vollständige Probleme?

Satz 5.12 *D-REACHABILITY ist* NL*-vollständig.*

Beweis: Wir haben bereits gesehen, dass D-REACHABILITY in NL liegt. Es ist also $L \leq_{\text{LOG}}$ D-REACHABILITY für eine beliebige Sprache $L \in \mathtt{NL}$ nachzuweisen.

Da $L \in \mathtt{NL}$, gibt es eine nichtdeterministische Turingmaschine M, die L mit logarithmischem Speicherbedarf erkennt. Für eine Eingabe w betrachten wir den *Berechnungsgraph* $G_M(w)$ von M auf Eingabe w. Die Konfigurationen bilden die Knoten von $G_M(w)$. Wir fügen eine Kante von Konfiguration c nach Konfiguration d ein, wenn M auf Eingabe w in einem Schritt von c nach d gelangen kann. Zusätzlich fügen wir eine Kante von jeder akzeptierenden Haltekonfiguration zu einem neuen Knoten ein, dem wir den „Namen“ 2 geben. Der der Anfangskonfiguration entsprechende Knoten erhält den Namen 1.

Behauptung: $G_M(w)$ kann von einer deterministischen I-O Turingmaschine mit logarithmischem Speicherplatz berechnet werden.

Beweis: Wir beachten zuerst, dass $G_M(w)$ nur polynomiell (in $|w|$) viele Knoten besitzt, da die Konfigurationenzahl polynomiell ist. Wir müssen also auf logarithmischem Platz die polynomiell große Adjazenzmatrix von $G_M(w)$ berechnen. Diese Aufgabe ist aber einfach, da wir ja nur die Möglichkeit eines Ein-Schritt Übergangs zwischen zwei Konfigurationen (zu jeweils logarithmischem Speicherplatz) überprüfen müssen. □

Wir weisen $G_M(w)$ der Eingabe w zu und erhalten

$$w \in L \Leftrightarrow G_M(w) \in \text{ D-REACHABILITY}.$$

Die Behauptung des Satzes folgt. □

Das nächste Ergebnis zeigt, dass NL in P enthalten ist. Dieses Ergebnis ist nicht mehr überraschend, sondern wegen der NL-Vollständigkeit des Problems D-REACHABILITY naheliegend.

Satz 5.13 $\mathtt{DL} \subseteq \mathtt{NL} \subseteq \mathtt{P} \subseteq \mathtt{NP}$.

Beweis: Die Beziehungen $\mathtt{DL} \subseteq \mathtt{NL}$ sowie $\mathtt{P} \subseteq \mathtt{NP}$ sind offensichtlich und es genügt der Nachweis von $\mathtt{NL} \subseteq \mathtt{P}$.

Man überzeuge sich zuerst, dass aus $L \leq_{\text{LOG}} K$ und $K \in \mathtt{P}$ auch $L \in \mathtt{P}$ folgt. (Wende Satz 5.8 (a) an!) Für eine beliebige Sprache L in NL gilt $L \leq_{\text{LOG}}$ D-REACHABILITY. Da D-REACHABILITY in P liegt, folgt somit auch $L \in \mathtt{P}$. □

Es könnte durchaus sein, dass beide Inklusionen in $\mathtt{DL} \subseteq \mathtt{NL} \subseteq \mathtt{P}$ echte Inklusionen sind.

Beispiel 5.2 Die Klasse LOGCFL besteht aus allen Entscheidungsproblemen, die LOGSPACE-reduzierbar auf eine kontextfreie Sprache sind.

Man kann zum Beispiel zeigen, dass NL eine Teilklasse von LOGCFL ist: Es gibt also eine kontextfreie Sprache, auf die D-REACHABILITY reduziert werden kann.

Aufgabe 81
Zeige, dass es eine kontextfreie Sprache L mit

$$\text{D} - \text{REACHABILITY} \leq_{LOG} L$$

gibt.

Desweiteren stimmt LOGCFL mit der Klasse aller Entscheidungsprobleme überein, die durch Schaltkreise logarithmischer Tiefe gelöst werden können. Während verlangt wird, dass der Fanin eines UND-Gatters zwei ist, ist der Fanin von ODER-Gattern unbeschränkt. Zudem muss man die Schaltkreise durch „einfache Algorithmen" beschreiben können –man sagt, dass die Schaltkreise uniform sind.

5.3.3 Der Satz von Savitch

Um wieviel mächtiger ist NL im Vergleich zu DL?

Aufgabe 82
Zeige: Wenn $\texttt{Dspace}(\log n) = \texttt{Nspace}(\log n)$, dann gilt $\texttt{Dspace}(s) = \texttt{Nspace}(s)$ für alle platz-konstruierbaren Funktionen s.

Aufgabe 83
Es seien s, S Funktionen mit $s = O(S)$ und $s = \Omega(n)$. Weiterhin gelte für alle g mit $g = \Omega(s)$ und $g = O(S)$, dass g nicht platz-konstruierbar ist. Zeige: Dann gilt $\texttt{Dspace}(s) = \texttt{Dspace}(S)$.

Hinweis: Zeige, dass die Funktion $\text{space}_M(n)$ für Turingmaschinen M mit mindestens linearem Platzverbrauch platz-konstruierbar ist.

Satz 5.14 *(Der Satz von Savitch)*

(a) *D-REACHABILITY* $\in \texttt{Dspace}(\log_2^2 n)$.

(b) *Die Funktion s sei platz-konstruierbar. Dann ist*

$$\texttt{Nspace}(s) \subseteq \texttt{Dspace}(s^2)$$

und insbesondere folgt

$$\texttt{NL} \subseteq \texttt{Dspace}(\log_2^2 n) \quad \textit{und} \quad \texttt{PSPACE} = \texttt{NPSPACE}.$$

Beweis (a): Leider können wir D-REACHABILITY weder mit Tiefensuche noch mit Breitensuche lösen, denn sowohl der Stack der Tiefensuche wie auch die Queue der Breitensuche verlangen bis zu linearem Speicherplatz. Wir beschreiben deshalb ein neues Traversierungsverfahren, das wesentlich speicher-effizienter sein wird.

Algorithmus 5.2 *(Eine platz-effiziente Traversierung)*

(1) *Der Graph G sei als Adjazenzmatrix gegeben. G bestehe aus n Knoten.*

(2) *Für jedes $m \leq n - 1$ rufe Algorithmus 5.3 mit dem Eingabegraphen G sowie den Parametern 1, 2 und m auf.*

/ Algorithmus 5.3 wird überprüfen, ob es in G einen Weg der Länge m ($m \leq n-1$) vom Knoten* 1 *zum Knoten 2 gibt. */*

(3) Akzeptiere genau dann, wenn Algorithmus 5.3 mindestens einmal akzeptiert.

Algorithmus 5.3

(1) Die Eingaben seien der Graph G sowie die Knoten u und v und die Weglänge m.

(2) Wenn m=1, dann akzeptiere, falls (u, v) eine Kante von G ist und verwerfe ansonsten.

(3) Für alle Knoten w führe zwei rekursive Aufrufe mit den jeweiligen Parametern u, w und $\lceil \frac{m}{2} \rceil$ beziehungsweise w, v und $\lfloor \frac{m}{2} \rfloor$ durch.

(4) Akzeptiere, wenn es einen Knoten w mit zwei akzeptierenden Aufrufen gibt und verwerfe ansonsten.

Algorithmus 5.2 akzeptiert offensichtlich genau dann, wenn es einen Weg vom Knoten 1 zum Knoten 2 gibt. Die entscheidende Frage ist die Größe des Speicherplatzbedarfs. Der rekursive Algorithmus 5.3 benötigt einen Stack der Höhe $\log_2 n$, wobei jedes Element des Stacks einem Knoten entspricht und damit ebenfalls logarithmischen Speicher in Anspruch nimmt. Insgesamt benötigen wir, wie versprochen, also Speicherplatz $O(\log_2^2 n)$.

Am Rande sei vermerkt, dass Algorithmus 5.3 zwar speicher-effizienter als Tiefen- oder Breitensuche ist, dafür aber bis zu $2^{O(\log_2^2 n)}$ Schritte verschlingt.

(b) Es sei $s(n) \geq \log_2 n$ und s sei platz-konstruierbar. Weiterhin sei M eine beliebige nichtdeterministische Turingmaschine mit Speicherplatzbedarf s und w sei eine Eingabe. Wir konstruieren eine deterministische Turingmaschine M^*, die M auf Speicherplatz $O(s^2)$ simuliert.

Da s platz-konstruierbar ist, kann M^* einen Speicherplatz von $s(n)$ Zellen abstecken und damit die Konfigurationen von M systematisch erzeugen. Wir gehen davon aus, dass die Startkonfiguration den Namen 1 trägt und dass es genau eine akzeptierende Haltekonfiguration gibt, der wir den Namen 2 zuweisen. M^* wendet nun Algorithmus 5.2 auf den Berechnungsgraphen $G_M(w)$ (mit höchstens $2^{O(s(|w|))}$ Knoten) an und akzeptiert genau dann, wenn Algorithmus 5.2 akzeptiert.

Offensichtlich akzeptiert M^* die Eingabe w genau dann, wenn M die Eingabe w akzeptiert. Die Behauptung folgt, da M^* nur den Speicherplatzbedarf $O(s^2)$ hat: Der Name eines jeden Knotens hat höchstens $O(s)$ Bits, und die Rekursionstiefe ist ebenfalls durch $O(s)$ beschränkt. □

Der Beweis des Satzes von Savitch betont die überragende Rolle von D-REACHABILITY für die Speicherplatzkomplexität. Aussagen über die nichtdeterministische Speicherplatzkomplexität werden zuerst für `NL` (und damit für D-REACHABILITY) nachgewiesen. Sodann ist zu gewährleisten, dass sich die Aussagen nach „oben" vererben, also auch für größeren Speicherplatz gelten.

Aufgabe 84
Jede Sprache L in NP kann wie folgt beschrieben werden: Es gibt eine Sprache $K \in P$ und eine Konstante c, so dass für alle Eingaben x

$$x \in L \iff \exists y \in \{0,1\}^* (|y| \leq |x|^c \wedge (x, y) \in K).$$

Definiere $\Sigma_1^p = NP$, $\Pi_1^p = co - NP$ sowie für alle $k \in \mathbb{N}, k > 1$

$$\Sigma_k^p = \{L \subseteq \{0,1\}^* \quad | \quad \exists K \in \Pi_{k-1}^p, c \in \mathbb{N} \forall x \in \{0,1\}^* (x \in L \iff \exists y \in \{0,1\}^* (|y| \leq |x|^c \wedge (x, y) \in K))\}.$$

Weiterhin ist $\Pi_k^p = \text{co}\Sigma_k^p$ und $PH = \cup_{k \in \mathbb{N}} \Sigma_k^p$. PH heißt die polynomielle Hierarchie.

Zeige: (a) `NP` = `coNP` $\iff PH = NP$.

(b) $PH \subseteq$ `PSPACE`.

5.3.4 Der Satz von Immerman und Szlepscenyi

Wir zeigen als nächstes, dass überraschenderweise jede Sprache und ihr Komplement die gleichen nichtdeterministischen Speicherplatz-Ressourcen benötigen, oder in anderen Worten, dass nichtdeterministischer Speicherplatz abgeschlossen ist unter Komplementbildung. Der wesentliche Schritt wird der Nachweis sein, dass das Komplement von D-REACHABILITY, also

$$\text{D-UNREACHABILITY} = \{G \mid \text{es gibt keinen Weg von Knoten 1 nach Knoten 2}\}$$

ebenfalls in $\texttt{NL}$ liegt.

Satz 5.15 *(Der Satz von Immerman und Szlepscenyi)*

(a) D-UNREACHABILITY $\in \texttt{NL}$.

(b) Die Funktion s sei platz-konstruierbar. Dann ist

$$\texttt{Nspace}(s) = co\texttt{Nspace}(s),$$

wobei $co\texttt{Nspace}(s) = \{\overline{L} \mid L \in \texttt{Nspace}(s)\}$ *genau aus den Komplementen von Sprachen aus* $\texttt{Nspace}(s)$ *besteht.*

Beweis (a): Der Graph G sei die Eingabe für D-UNREACHABILITY. *Angenommen*, wir könnten das Anzahlproblem in $\texttt{NL}$ lösen, also die Anzahl m der von Knoten 1 aus erreichbaren Knoten bestimmen. Wir zeigen zuerst, dass dann auch D-UNREACHABILITY zu $\texttt{NL}$ gehört. Im zweiten Schritt lösen wir dann das Anzahlproblem mit nichtdeterministisch logarithmischem Platz.

Wir nehmen also an, dass wir die Zahl m der von 1 aus erreichbaren Knoten kennen. Unsere nichtdeterministische Maschine versucht jetzt nacheinander m verschiedene Knoten $v_1, \ldots, v_m$ zu finden, die von 1 aus erreichbar sind. Wenn der Knoten 2 von all diesen Knoten verschieden ist, dann kann M folgern, dass 2 nicht von 1 aus erreichbar ist und M wird akzeptieren.

Im Detail sieht das Vorgehen von M wie folgt aus. Die erste Phase von M ist trivial: M „setzt" $v_1 = 1$, da 1 von 1 aus erreichbar ist. In der $i+1$ten Phase rät M einen Knoten v_{i+1} und rät sukzessive eine Knotenfolge beginnend mit Knoten 1. M verwirft, wenn $v_{i+1} \leq v_i$ oder wenn $v_{i+1} = 2$ oder wenn die geratene Knotenfolge keinem im Knoten v_{i+1} endenden Weg entspricht. Ansonsten akzeptiert M, falls $i+1 = m$, beziehungsweise beginnt Phase $i+1$, falls $i+1 < m$.

Wir haben also D-UNREACHABILITY gelöst, wenn das Anzahlproblem gelöst ist. Wie können wir aber das Anzahlproblem in $\texttt{NL}$ lösen? Sei m_i die Anzahl der Knoten, die durch Wege der Länge höchstens i von Knoten 1 aus erreichbar sind. Offensichtlich ist $m_0 = 1$ und $m_{n-1} = m$. Wir müssen nur für jedes i zeigen, dass m_{i+1} in $\texttt{NL}$ berechnet werden kann, *wenn* m_i bekannt ist.

Wir können also annehmen, dass m_i bekannt ist. Wir setzen zu Anfang $m_{i+1} = 1$ und wiederholen dann im wesentlichen das Vorgehen von M für alle Knoten (in aufsteigender Reihenfolge). Wenn Knoten k behandelt wird, raten wir sukzessive Knoten $v_{i,1} < \cdots < v_{i,m_i}$ und verifizieren, dass jeder geratene Knoten vom Knoten 1 durch einen Weg der Länge höchstens i erreichbar ist.

Wenn wir für den gegenwärtigen Knoten k gerade den Knoten $v_{i,r}$ verifiziert haben, dann prüfen wir zusätzlich nach, ob $v_{i,r}$ ein Vorgänger von k ist. Ist dies der Fall, dann erhöhen wir m_{i+1} um 1 und brechen die Behandlung von Knoten k ab, um mit der Behandlung von Knoten $k+1$ zu beginnen. Ist dies nicht der Fall, dann setzen wir die Ratephase für Knoten mit Abstand

höchstens i vom Knoten 1 fort. In diesem Fall beginnt die Behandlung von Knoten $k+1$ erst, wenn alle m_i Knoten überprüft wurden.

Also ist auch das Anzahlproblem in NL lösbar und die Behauptung ist gezeigt.

(b) Die Sprache L werde von einer nichtdeterministischen Turingmaschine M mit Speicherplatzbedarf s erkannt. Wir wissen, dass s platz-konstruierbar ist und damit gilt insbesondere auch $s \geq \log_2 n$. Unser Ziel ist die Konstruktion einer nichtdeterministischen Turingmaschine M^*, die das Komplement $\overline{L}$ mit Speicherplatzbedarf $O(s)$ erkennt. M^* muss also nachprüfen, ob M für eine Eingabe w eine akzeptierende Berechnung besitzt (sprich: einen Weg von der Startkonfiguration 1 zur akzeptierenden Haltekonfiguration 2 besitzt) und genau dann akzeptieren, wenn es eine akzeptierende Berechnung nicht gibt.

M^* wendet den Algorithmus aus Teil (a) auf den Berechnungsgraphen $G_M(w)$ an, wobei M^* die Existenz von Kanten selbst klären muss. Offensichtlich gelingt diese Klärung in Platz $O(s)$, da die Konfigurationen von M nur Platz $O(s)$ benötigen. $\square$

Aufgabe 85
Zeige: Wenn L vollständig für NL ist, dann ist auch das Komplement $\overline{L}$ vollständig für NL.

Die Sprache 2-SAT besteht aus allen erfüllbaren aussagenlogischen Formeln in konjunktiver Normalform mit jeweils höchstens zwei Literalen pro Klausel.

Korollar 5.16 *2-SAT ist NL-vollständig.*

Beweis: Wir zeigen zuerst die nicht-offensichtliche Aussage, dass 2-SAT in NL liegt. Sei also ϕ eine Formel in konjunktiver Normalform mit jeweils zwei Literalen pro Klausel. Wir weisen ϕ den gerichteten Graphen $G(\phi)$ zu, wobei die Knoten von $G(\phi)$ den Literalen von ϕ entsprechen. Wir setzen eine Kante vom Literal α zum Literal β ein, falls $\neg\alpha \vee \beta$ eine Klausel ist. Eine Kante (α, β) entspricht also einer Implikation $\alpha \rightarrow \beta$. Für jede Kante (α, β) fügen wir auch die „symmetrische" Kante $(\neg\beta, \neg\alpha)$ in $G(\phi)$ ein; beachte, dass $\alpha \rightarrow \beta$ und $\neg\beta \rightarrow \neg\alpha$ logisch äquivalent sind.

Behauptung: ϕ ist genau dann nicht erfüllbar, wenn es ein Literal x gibt, so dass es einen Weg von x nach $\neg x$ wie auch einen Weg von $\neg x$ nach x gibt.

Beweis $\Leftarrow$: Die Existenz eines Weges von α nach β erzwingt für jede erfüllende Belegung, die α auf 1 setzt, auch dass β auf 1 gesetzt wird. Also müssen x und $\neg x$ auf denselben Wert gesetzt werden, was aber nicht erlaubt ist.

$\Rightarrow$: Wir wissen also, dass ϕ nicht erfüllbar ist. Wir nehmen an, dass die Behauptung falsch ist und konstruieren dann eine erfüllende Belegung. Wir beginnen mit irgendeinem ungesetzten Literal x, für das wir annehmen können, dass es keinen Weg von x nach $\neg x$ gibt. Wir weisen x und allen von x in $G(\phi)$ erreichbaren Literalen den Wert 1 zu und weisen $\neg x$ und allen Literalen, die $\neg x$ erreichen, den Wert 0 zu.

Unser Vorgehen ist wohl-definiert, da es kein Literal y geben kann, so dass sowohl y wie auch $\neg y$ von x aus erreichbar sind. Der Graph $G(\phi)$ garantiert ja mit seinen symmetrischen Kanten auch symmetrische Wege: Zu jedem Weg von x nach y gibt es auch einen Weg von $\neg y$ nach $\neg x$ und simultane Wege von x nach y wie auch von x nach $\neg y$ implizieren einen (ausgeschlossenen) Weg von x nach $\neg x$ über $\neg y$.

Weiterhin kann es kein von x aus erreichbares Literal geben, das vorher auf 0 gesetzt wurde. Ein solcher Fall ist ausgeschlossen, da dann x schon gesetzt worden wäre.

Offensichtlich können wir unser Vorgehen solange wiederholen bis alle Literale gesetzt sind. Wir haben aber alle Klauseln erfüllt und die Formel ϕ ist im Widerspruch zur Annahme erfüllt. □

Aus der Behauptung erhalten wir sofort, dass die Nicht-Erfüllbarkeit in NL liegt, da wir ja nur die Existenz von Wegen zwischen sich widersprechenden Literalen zu raten brauchen. Damit folgt aber aus dem Satz von Immerman und Szlepscenyi, dass Erfüllbarkeit, also 2-SAT, in NL liegt.

Es bleibt zu zeigen, dass 2-SAT NL-vollständig ist. Wir konstruieren zuerst die Reduktion D-UNREACHABILITY $\leq_{LOG}$ 2-SAT. Sei also G ein Eingabegraph. Wir weisen dem Graphen G eine Formel ϕ_G wie folgt zu. Wir fassen die Knoten von G als Variablen auf und fügen für jede Kante (u, v) die Klausel $\neg u \vee v$ ein. Desweiteren verwenden wir die Einer-Klauseln $1'$ sowie $\neg 2'$.

Behauptung: ϕ_G ist genau dann erfüllbar, wenn es keinen Weg von 1' nach 2' gibt.

Beweis: Wenn es keinen Weg von 1' nach 2' gibt, dann ist ϕ_G erfüllt, wenn wir 1' und allen von 1' erreichbaren Knoten den Wert 1 und den restlichen Knoten den Wert 0 zuweisen. Gibt es hingegen einen Weg von 1' nach 2', dann müssen alle Knoten des Weges (und damit auch 2') auf 1 gesetzt werden: ϕ_G ist nicht erfüllbar. □

Die Transformation $G \to \phi_G$ etabliert also die Reduktion. Jetzt brauchen wir nur noch zu beachten, dass D-UNREACHABILITY NL-vollständig ist (warum?) und das Ergebnis folgt mit Korollar 5.18. □

Wir kennen also mittlerweile vier NL-vollständige Probleme, nämlich D-REACHABILITY und 2SAT sowie die beiden Komplemente. Bipartitness und das Wortproblem für NFAs sind auch NL-vollständig:

Aufgabe 86
Wir definieren die Sprache $L = \{ \langle M \rangle w \mid M \text{ ist ein NFA und } M \text{ akzeptiert } w\}$.

Zeige, dass L NL-vollständig ist.

5.4 PSPACE-Vollständigkeit

Wir möchten die schwierigsten Sprachen in PSPACE bezüglich der polynomiellen Reduktion bestimmen. Zur Erinnerung: Wir sagen, dass ein Entscheidungsproblem L_1 genau dann auf ein Entscheidungsproblem L_2 polynomiell reduzierbar ist (geschrieben $L_1 \leq_P L_2$), wenn

$$w \in L_1 \Leftrightarrow T(w) \in L_2$$

für alle Eingaben w von L_1 gilt. Die Transformation $w \mapsto T(w)$ muss in polynomieller Zeit durch eine deterministische Turingmaschine berechenbar sein.

Die Definition der PSPACE-vollständigen Sprachen folgt dem Schema der NL- und NP-Vollständigkeit.

Definition 5.17 *Sei K eine Sprache.*

(a) *K heißt* PSPACE*-hart, falls $L \leq_P K$ für jede Sprache $L \in$* PSPACE *gilt.*

(b) *K heißt* PSPACE*-vollständig, falls K* PSPACE*-hart ist und falls*
$K \in$ PSPACE.

PSPACE-harte Sprachen sind mindestens so schwierig wie NP-harte Sprachen:

Aufgabe 87
Zeige: Wenn die Sprache L PSPACE-hart ist, dann ist L auch NP-hart.

Wie auch im Fall der NP- oder NL-Vollständigkeit führt eine Berechnung der vollständigen Probleme innerhalb der kleineren Klasse (sprich: P) zum Kollaps der größeren Klasse. Weiterhin genügt, wie üblich, zum Nachweis der Vollständigkeit die Reduktion auf ein vollständiges Problem.

Korollar 5.18 *(PSPACE-vollständige und PSPACE-harte Sprachen)*

(a) Die Sprache K sei PSPACE-vollständig. Dann gilt

$$K \in \mathtt{P} \;\Leftrightarrow\; \mathtt{P} = \mathtt{PSPACE}.$$

(b) Die Sprache K sei PSPACE-hart. Wenn $K \leq_P L$, dann ist auch L PSPACE-hart.

5.4.1 QBF: Quantifizierte Boolesche Formeln

Wir konstruieren die Sprache QBF, die sich später als PSPACE-vollständig herausstellen wird. Die Worte in QBF entsprechen Kodierungen $\langle\phi\rangle$ von *quantifizierten Booleschen Formeln* ϕ. Die Formel ϕ besteht aus einem Quantorenteil gefolgt von einer aussagenlogischen Formel α. Der Quantorenteil besteht aus All- und Existenz-Quantoren, so dass jede in α vorkommende Variable von genau einem Quantor gebunden wird. Wir definieren

$$\text{QBF} \;= \{\langle\phi\rangle \mid \phi \text{ ist eine wahre quantifizierte Boolesche Formel }\}.$$

Beispiel 5.3 Die Formel $\phi \equiv \exists p \forall q((p \vee \neg q) \wedge (\neg p \vee q))$ ist falsch, denn sie drückt die Äquivalenz von p und q aus. Sicherlich gibt es aber keinen Wahrheitswert für p, der mit den beiden Wahrheitswerten 0 und 1 äquivalent ist. Somit ist $\phi \notin$ QBF.

Die Formel $\psi \equiv \forall p \exists q((p \vee \neg q) \wedge (\neg p \vee q))$ ist hingegen wahr, denn zu jedem Wahrheitswert für p gibt es einen äquivalenten Wahrheitswert für q. Also ist $\psi \in$ QBF.

Satz 5.19 *QBF ist PSPACE-vollständig.*

Beweis: Wir zeigen zuerst, dass QBF in PSPACE liegt. Dazu betrachten wir den folgenden rekursiven Algorithmus.

Algorithmus 5.4 *(Erkennen wahrer quantifizierter Formeln auf polynomiellem Platz)*

(1) Die quantifizierte Booleschen Formel ϕ sei die Eingabe.

(2) Wenn ϕ keine Quantoren besitzt, dann besteht ϕ nur aus aussagenlogischen Verknüpfungen der Konstanten 0 und 1 und kann direkt ausgewertet werden. Gib die Auswertung aus.

(3) Wenn $\phi \equiv \exists p \psi(p)$, dann führe rekursive Aufrufe mit den quantifizierten Booleschen Formeln $\psi(0)$ und $\psi(1)$ durch. Wenn eine der Formeln zu 1 auswertet, dann gib 1 als Auswertung aus. Ansonsten gib 0 als Auswertung aus.

(4) *Wenn $\phi \equiv \forall p \psi(p)$, dann führe rekursive Aufrufe mit den quantifizierten Booleschen Formeln $\psi(0)$ und $\psi(1)$ durch. Wenn eine der Formeln zu 0 auswertet, dann gib 0 als Auswertung aus. Ansonsten gib 1 als Auswertung aus.*

Algorithmus 5.4 hat eine höchstens lineare Rekursionstiefe. In jedem Rekursionsschritt ist aber nur ein Wahrheitswert abzuspeichern, so dass der Algorithmus nur einen linearen Speicherplatzbedarf hat und QBF $\in$ PSPACE folgt.

Wir kommen zum Nachweis der PSPACE-Härte. Die Sprache $L \in$ PSPACE werde von deterministischen Turingmaschine M mit Speicherplatzbedarf $O(n^k)$ berechnet. Wir müssen für jede Eingabe w von L in polynomieller Zeit eine quantifizierte Boolesche Formel ϕ_w konstruieren, so dass

$$w \in L \ \Leftrightarrow \phi_w \text{ ist wahr}$$

gilt. Wir erinnern an den NP-Vollständigkeitsbeweis von KNF-SAT. Eine (nicht-quantifizierte) Boolesche Formel α_w^t kodiert die Konfiguration der Turingmaschine zum Zeitpunkt t. Die Kodierung gelingt durch Einbeziehung der aussagenlogischen Variablen

- $Kopf^t(z)$ für die Kopfposition. $Kopf^t(z)$ *soll* genau dann wahr ist, wenn der Kopf zum Zeitpunkt t auf Zelle z steht,

- $Zelle^t(z, a)$ für den Zelleninhalt. $Zelle^t(z, a)$ *soll* genau dann wahr ist, wenn die Zelle z zum Zeitpunkt t mit dem Buchstaben a beschriftet ist und

- $Zustand^t(q)$ für den aktuellen Zustand. $Zustand^t(q)$ *soll* genau dann wahr ist, wenn q der Zustand zum Zeitpunkt ist.

Weiterhin wird die beabsichtigte Bedeutung erzwungen, in dem durch die Konjunktion von zusätzlichen Klauseln sichergestellt wird, dass die Zelleninhalte, die Kopfbewegung und der neue Zustand sich gemäss der Arbeitsweise der Maschine verändern.

Wir kehren zurück zum PSPACE-Vollständigkeitsbeweis von QBF und müssen eine *kurze* Formel ϕ_w für die möglicherweise exponentiell lange Berechnung der Turingmaschine M zu schreiben! Sei T die kleinste Zweierpotenz, die größer als die Anzahl der Konfigurationen von M ist. c_0 ist die Anfangskonfiguration und c_a die eindeutig bestimmte akzeptierende Haltekonfiguration von M. (Wieso kann man verlangen, dass es genau eine akzeptierende Haltekonfiguration gibt?) Unser Ziel ist die Konstruktion von höchstens polynomiell langen Formeln $\psi_t(c, d)$, die genau dann wahr sein *sollen*, wenn M –in der Konfiguration c startend– die Konfiguration d nach höchstens t Schritten erreicht. Ist dies geschafft, dann setzen wir $\phi_w \equiv \psi_T(c_0, c_a)$ und ϕ_w ist genau dann wahr, wenn M die Eingabe w akzeptiert.

Die Formeln $\psi_1(c, d)$ sind einfach zu konstruieren. Wir müssen ausdrücken, dass $c = d$ oder dass d die Nachfolgekonfiguration von c ist. Wir übernehmen die Konfigurations-Kodierung durch die drei Typen der aussagenlogischen Variablen aus dem NP-Vollständigkeitsbeweis für KNF-SAT. Im Fall $c = d$ müssen wir nur die Äquivalenz der c- und d-Variablen fordern, während im anderen Fall die d-Variablen als Aktualisierung der c-Variablen zu formulieren sind. In jedem der beiden Fälle sind keinerlei Quantoren notwendig.

Wir benutzen Existenz-Quantoren in der rekursiven Definition von $\psi_t(c, d)$, um eine Zwischenkonfiguration e zu raten, sowie All-Quantoren, um die Formellänge klein zu halten. Wir setzen

$$\psi_{2t}(c, d) \ \equiv \ \exists e \forall f \forall g(\ ((f = c \wedge g = e) \vee (f = e \wedge g = d)) \rightarrow \psi_t(f, g) \ ).$$

Beachte, dass $\exists e$ einer Reihe von Existenz-Quantoren entspricht, nämlich den Existenz-Quantoren zu den Variablen der Konfiguration e bezüglich Kopfposition, Zelleninhalt und Zustand; die gleiche Aussage gilt analog für $\forall f$ und $\forall g$. Die Formel $\psi_{2t}(c, d)$ drückt aus, dass eine Berechnung

der Länge höchstens $2t$ aufgespalten werden kann in zwei aufeinanderfolgende Berechnungen der Länge höchstens t. Der All-Quantor erlaubt eine simultane Überprüfung der beiden Berechnungen von c nach e und von e nach d. Dementsprechend wächst die Formellänge additiv um höchstens $O(n^k)$, also höchstens um den Speicherplatzbedarf von M, und wir erhalten $O(n^{2k})$ als obere Schranke für die Länge der Formel $\psi_T(c_0, c_a)$. □

Wir können die Sprache QBF ein wenig vereinfachen ohne die PSPACE-Vollständigkeit zu verlieren. Insbesondere betrachten wir nur quantifizierte Boolesche Formeln ϕ, deren Quantoren strikt alternieren: auf jeden All-Quantor folgt also ein Existenz-Quantor auf den wiederum ein All-Quantor folgen muss. Diese Einschränkung ist oberflächlich, da wir stets beliebige Quantoren einfügen können, die in der Formel nicht vorkommende Variablen binden. In der zweiten und letzten Einschränkung fordern wir, dass sich die in ϕ quantifizierte aussagenlogische Formel in konjunktiver Normalform befindet. Wir definieren dann QBF* als die Menge aller quantifizierten Booleschen Formeln in QBF, die die beiden obigen Einschränkungen erfüllen.

Korollar 5.20 *QBF* ist* PSPACE*-vollständig.*

Beweis: Die Formel ϕ sei eine (nicht eingeschränkte) quantifizierte Boolesche Formel. Wir überführen ϕ in polynomieller Zeit in eine äquivalente, aber eingeschränkte Formel ϕ^*. Zuerst führen wir Dummy-Quantoren ein, um strikte Quantoren-Alternation zu sichern. Dann werden wir die aussagenlogische Formel α von ϕ durch Einführung neuer Variablen x in eine äquivalente Formel $\exists x \alpha^*(x)$ überführen, wobei α^* in konjunktiver Normalform sein wird.

Wir nehmen zuerst an, dass sich Negationen nur auf Variablen beziehen; ist dies nicht der Fall, dann schieben wir Negationen mit den de Morgan'schen Regeln bis zu den Variablen vor. Sodann gehen wir indukiv vor.

Fall 1: $\alpha = \alpha_1 \vee \alpha_2$. Wir erfinden neue Variablen x_1 und x_2. x_1 (bzw x_2) wird mit jeder Klausel der konjunktiven Normalform von α_1 (bzw. α_2) „verodert". Sei α^* die aus den beiden modifizierten konjunktiven Normalformen sowie der neuen Klausel $(\neg x_1 \vee \neg x_2)$ resultierende konjunkive Normalform. Beachte, dass $\alpha \leftrightarrow \exists x_1 \exists x_2 \alpha^*$.

Fall 2: $\alpha = \alpha_1 \wedge \alpha_2$. Wir definieren α^* als die Konjunktion der konjunktiven Normalform für α_1 mit der konjunktiven Normalform für α_2. Etwaige Existenz-Quantoren in α_1 oder α_2 sind nach vorne zu ziehen.

Wir können die Formel α^* in polynomieller Zeit konstruieren und die Behauptung folgt. □

Das PSPACE-vollständige Problem QBF^* hat also die Form $\exists x_1 \forall x_2 \exists x_3 \ldots \forall x_n P(x_1, \ldots, x_n)$ für ein KNF-Prädikat P. Eine solche Struktur läßt sich als ein Zweipersonen-Spiel auffassen:

> Der ziehende Spieler muss für eine gegebene Spielsituation einen Zug bestimmen, der gegen alle Züge des Gegenspielers in eine gewinnbare Spielsituation führt. Das Prädikat P definiert das Spiel.

In vielen interessanten Spielen ist die Spielauswertung P als eine aussagenlogische Formel darstellbar, und die Frage nach einer Gewinnstrategie für den ziehenden Spieler liegt dann in PSPACE. Tatsächlich kann man zeigen, dass viele Spiele (zum Beispiel $n \times n$ Versionen von Go oder Schach) sogar PSPACE vollständig sind, und dass daher Gewinnstrategien nicht in P berechenbar sind, es sei denn, es gilt P = PSPACE.

5.4.2 Das Geographie-Spiel

Wir definieren das GEOGRAPHIE-Spiel, eine Verallgemeinerung des Spiels, bei dem zwei Spieler abwechselnd noch nicht genannte Städtenamen wählen, wobei jede Stadt mit dem Endbuchstaben der zuvor genannten Stadt beginnen muß.

- **Die Eingabe:** Ein gerichteter Graph $G = (V, E)$ und ein ausgezeichneter Knoten $s \in V$.
- **Die Spielregeln:**
  – Zwei Spieler A und B wählen abwechselnd jeweils eine noch nicht benutzte Kante aus E.
  – Spieler A fängt an und wählt eine Kante mit Startknoten s.
  – Jede anschließend gewählte Kante muß im Endknoten der zuvor gewählten Kante beginnen.
  – Der Spieler, der als erster keine solche unbenutzte Kante mehr findet, verliert das Spiel.
- **Die Aufgabe:** Es ist zu entscheiden, ob ein optimal spielender Spieler A gegen jeden Spieler B auf G gewinnen kann.

Aufgabe 88
Zeige, dass GEOGRAPHIE durch eine deterministische Turingmaschine mit höchstens polynomiellem Speicherplatz gelöst werden kann. Warum gehört GEOGRAPHIE wahrscheinlich nicht zur Klasse NP?

5.4.3 NFA's und reguläre Ausdrücke

Im Entscheidungsproblem der Nicht-Universalität für reguläre Ausdrücke ist ein regulärer Ausdruck R genau dann zu akzeptieren, wenn $L(R) \neq \Sigma^*$ für die von R beschriebene Sprache $L(R)$ gilt. Ein solch einfaches Problem kann doch nicht schwierig sein, oder? Tatsächlich ist die Nicht-Universalität für reguläre Ausdrücke hammer-hart, und als Konsequenz kann die Größe von minimalen regulären Ausdrücken oder minimalen NFA's nur völlig unbefriedigend approximiert werden.

Satz 5.21 *(a) Die Nicht-Universalität für reguläre Ausdrücke ist ebenso* PSPACE*-hart wie die Nicht-Universalität für NFA.*

(b) Das Äquivalenzproblem für reguläre Ausdrücke, also die Frage, ob zwei gegebene reguläre Ausdrücke R_1 und R_2 dieselbe Sprache beschreiben –also ob $L(R_1) = L(R_2)$ gilt–, ist PSPACE*-hart. Das Äquivalenzproblem für NFA ist ebenfalls* PSPACE*-hart.*

(c) Es gelte P $\neq$ PSPACE*. Dann ist es nicht möglich, für einen gegebenen regulären Ausdruck oder NFA A der Größe m, die Größe eines minimalen regulären Ausdrucks oder eines minimalen NFA's innerhalb des Faktors $o(m)$ effizient zu approximieren.*

Beweis (a): Das Entscheidungsproblem QBF ist PSPACE-vollständig und kann von einer deterministischen Turingmaschine

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \{q_f\})$$

gelöst werden, die in-place arbeitet, also nie den Eingabebereich verlässt.

Aufgabe 89
Wenn M eine nichtdeterministische Turingmaschine ist, die mit linearem Speicherplatz arbeitet, dann gibt es eine zu M äquivalente in-place Turingmaschine M'.

Zusätzlich können wir verlangen, dass M für jede Eingabe der Länge n mindestens 2^n Schritte benötigt.

Für eine Eingabe w für M konstruieren wir einen regulären Ausdruck R_w, der alle Worte akzeptiert, die *nicht* mit der Konfigurationenfolge einer akzeptierenden Berechnung von M auf Eingabe w übereinstimmen. Haben wir die Konstruktion von R_w in polynomieller Zeit geschafft, dann haben wir $L(M)$ –und damit QBF– auf die Nicht-Universalität für reguläre Ausdrücke reduziert und, wie gefordert, die `PSPACE`-Härte nachgewiesen. Warum?

$$\begin{aligned} w \in L(M) &\Leftrightarrow \text{ nur die Konfigurationenfolge der akzeptierenden Berechnung} \\ &\quad\;\; \text{von } M \text{ auf } w \text{ gehört nicht zu } L(R_w) \\ &\Leftrightarrow L(R_w) \neq \Sigma^*. \end{aligned}$$

Um Konfigurationen zu kodieren, benutzen wir das Alphabet

$$\Sigma' = Q \times \Gamma \cup \Gamma \cup \{\#\}.$$

Das neue Symbol $\#$ trennt Konfigurationen, ein Symbol $[q, a] \in Q \times \Sigma$ repräsentiert die Kopfposition von M und gibt an, dass gegenwärtig der Buchstabe a gelesen wird.

Wir stellen sicher, dass nur die Konfigurationenfolge einer akzeptierenden Berechnung von M verworfen wird, wenn

(1) die Anfangskonfiguration nicht von der Form

$$\#[q_0, w_1]w_2 \cdots w_n\#$$

ist oder

(2) keine Konfiguration der Konfigurationenfolge den Buchstaben $[q_f, \gamma]$ für irgendein $\gamma \in \Gamma$ enthält oder

(3) die Folge nicht mit dem Trennsymbol $\#$ endet oder

(4) wenn sich der Bandinhalt oder der Zustand zwischen aufeinanderfolgenden Konfigurationen auf eine nicht-legale Weise ändert.

Aufgabe 90
Konstruiere R_w als Vereinigung von vier regulären Ausdrücken der Länge $O(|w|)$–also einen Ausdruck für jeden der vier Fälle.

Hinweis: Um einen „kurzen“ regulären Ausdruck für den vierten Fall zu erhalten, beachte, dass in einer legalen Folge y von Konfigurationen für jedes Teilwort $y_{i-1}y_iy_{i+1}$ das „neue“ Symbol y_{i+n+1} eine Funktion von $y_{i-1}y_iy_{i+1}$ ist. Insbesondere, wenn x eine illegale Konfigurationenfolge ist, dann gilt $x_{i+n+1} \neq x_i$, obwohl der Kopf nicht auf Position i gestanden ist, oder x_{i+n+1} wird falsch aktualisiert.

Damit ist die Behauptung für reguläre Ausdrücke gezeigt. Das entsprechende Ergebnis für NFA folgt mit einem völlig analogem Argument.

(b) Die Nicht-Universalität für einen regulären Ausdruck R ist äquivalent zur Frage, ob $L(R) \neq L(\Sigma^*)$ gilt, d.h. ob die beiden regulären Ausdrücke R und Σ^* äquivalent sind. Die `PSPACE`-Härte

des Äquivalenzproblems für reguläre Ausdrücke folgt also aus Teil (a) und Gleiches gilt für die PSPACE-Härte des Äquivalenzproblems für NFA.

(c) Wir nehmen $\mathtt{P} \neq \mathtt{PSPACE}$ an. Nach Korollar 5.18 können dann PSPACE-vollständige Entscheidungsprobleme nicht effizient gelöst werden.

Insbesondere kann also nicht effizient bestimmt werden, ob $w \in L(M)$ für die Turingmaschine aus Teil (a) gilt. Nun ist $w \notin L(M)$ genau dann, wenn $L(R_w) = \Sigma^*$, bzw. genau dann, wenn der minimale, mit R_w äquivalente reguläre Ausdruck eine beschränkte Länge hat. Ist hingegen $w \in L(M)$, dann folgt $L(R_w) \neq \Sigma^*$ und genauer $L(R_w) = \Sigma^* \setminus \{y\}$ für die Konfigurationenfolge y der akzeptierenden Berechnung für Eingabe w. Wie lang muss ein minimaler, mit R_w äquivalenter Ausdruck in diesem Fall mindestens sein?

Wir haben gefordert, dass die Turingmaschine M mindestens 2^n Schritte für Eingaben der Länge n benötigt und deshalb hat y mindestens die Länge $2^{|w|}$. Ein deterministischer endlicher Automat benötigt mindestens $|y| \geq 2^{|w|}$ Zustände, um $\Sigma^* \setminus \{y\}$ zu akzeptieren, ein regulärer Ausdruck oder ein NFA muss deshalb mindestens $|w|$ Zustände besitzen. Aber R_w hat die Länge $O(|w|)$, und wir können deshalb nicht effizient unterscheiden, ob ein minimaler äquivalenter regulärer Ausdruck die Länge $O(1)$ oder $\Omega(|w|)$ besitzt.

Die Länge eines minimalen äquivalenten regulären Ausdrucks kann also so gut wie nicht nicht approximiert werden. Gleiches gilt aber auch für die Größe von äquivalenten minimalen NFA, denn auch ihre Größe variiert ziwschen $O(1)$ –falls $w \notin L(M)$– und $\Omega(|w|)$ –falls $w \in L(M)$. $\square$

Aufgabe 91
Gib einen Algorithmus an, der mit polynomiell beschränktem Platz arbeitet, und der bei Eingabe eines nichtdeterministischen endlichen Automaten einen nichtdeterministischen endlichen Automaten mit minimaler Zustandsanzahl für dieselbe Sprache konstruiert.

Hinweis: Gib einen nichtdeterministischen Algorithmus an. Zum Vergleich der Sprachen zweier nichtdeterministischer endlicher Automaten mit höchstens n Zuständen reicht es aus, das Verhalten auf allen Worten der Länge 2^n zu vergleichen.

Aufgabe 92
Es ist zu entscheiden, ob zwei nichtdeterministische endliche Automaten N_1 und N_2 äquivalent sind. Zeige, dass dieses Entscheidungsproblem PSPACE-vollständig und nicht nur PSPACE-hart ist.

5.5 Komplexitätsklassen und die Chomsky Hierarchie

Unser Ziel ist ein Vergleich der bereits betrachteten Komplexitätsklassen und Sprachenklassen

Wir haben bisher die Komplexitätsklassen DL, NL, P, NP und die Klasse PSPACE, der auf polynomiellen Speicherplatz entscheidbaren Sprachen kennengelernt und die Klassen der regulären und kontextfreien Sprachen untersucht. Insbesondere haben wir eingeschränkte Grammatiken als Grundlagen für die Beschreibung von Programmiersprachen behandelt: Die beiden vorrangigen Ziele einer Programmiersprache sind entgegengesetzt, nämlich

1. eine effiziente Lösung des *Wortproblems*. Es sollte in vertretbarer Zeit entscheidbar sein, ob ein vorgelegtes Wort ableitbar ist, bzw. ob ein Programm syntaktisch korrekt ist. Dauert dieser Entscheidungsvorgang zu lange, so kommt die Sprachenklasse als mögliche Grundlage für Programmiersprachen und Compiler nicht in Frage.

2. Die Sprachen sollten möglichst ausdruckstark sein, um höhere Programmiersprachen komfortabel darzustellen zu können.

Wir führen die Klasse der kontextsensitiven Grammatiken ein und erhalten damit die Chomsky-Hierarchie, eine Grobeinteilung der Sprachenklassen.

Definition 5.22 (Chomsky-Hierarchie)

(a) Grammatiken ohne jede Einschränkung heißen Typ-0 Grammatiken. *Die entsprechende Sprachenfamilie ist*

$$\mathcal{L}_0 = \Big\{ L(G) \,|\, G \textit{ ist vom Typ 0} \Big\}$$

(b) Eine Grammatik G mit Produktionen der Form

$$u \to v \quad \textit{mit } |u| \leq |v|$$

heißt Typ-1 *oder* kontextsensitiv. *Die zugehörige Sprachenfamilie ist*

$$\mathcal{L}_1 = \Big\{ L(G) \,|\, G \textit{ ist vom Typ 1} \Big\} \cup \Big\{ L(G) \cup \{\varepsilon\} \,|\, G \textit{ ist vom Typ 1} \Big\}$$

(c) Eine Grammatik G mit Produktionen der Form

$$u \to v \quad \textit{mit } u \in V \textit{ und } v \in (V \cup \Sigma)^*$$

heißt Typ-2 *oder* kontextfrei. *Die zugehörige Sprachenfamilie ist*

$$\mathcal{L}_2 = \Big\{ L(G) \,|\, G \textit{ hat Typ 2} \Big\}$$

(d) Eine reguläre Grammatik heißt auch Typ-3 Grammatik. *Die zugehörige Sprachenfamilie ist*

$$\mathcal{L}_3 = \Big\{ L(G) \,|\, G \textit{ hat Typ 3} \Big\}$$

Beispiel 5.4 Die kontextfreie Grammatik G mit Startsymbol S und Produktionen

$$S \to 0S0 \,|\, 1S1 \,|\, \varepsilon$$

erzeugt alle Palindrome über dem Alphabet $\{0, 1\}$. Diese Sprache ist offensichtlich nicht regulär. Da anderseits jede reguläre Grammatik kontextfrei ist, zeigt dieses Beispiel, dass $\mathcal{L}_2$ eine echte Obermenge von $\mathcal{L}_3$ ist.

Beispiel 5.5 Wir entwerfen eine kontextsensitive Grammatik, die die Sprache

$$K = \Big\{ a^i b^i c^i \,|\, i \geq 0 \Big\}$$

erkennt. Die Grammatik G besitzt drei Variablen, nämlich das Startsymbol S sowie die Variablen R und L. Die Produktionen haben die Form

$$\begin{array}{lcl} S & \to & \varepsilon \,|\, abc \\ S & \to & aRbc \end{array}$$

R wird ein neues a und b einführen, dann nach rechts wandern bis das erste c angetroffen wird. Ein c wird eingefügt, und R wird durch L ersetzt:

$$aRb \to aabbR, bRb \to bbR \text{ und } bRc \to bcc \,|\, Lbcc$$

Die Variable L läuft nach links bis das erste a gefunden wird. Dann wird L durch R ersetzt:

$$bL \to Lb \text{ und } aLb \to aRb.$$

Die Sprache Kist nicht kontextfrei. Da kontextfreie Sprachen L mit $\varepsilon \notin L$ durch Grammatiken in Chomsky Normalform erzeugt werden können, ist jede kontextfreie Sprache auch kontextsensitiv, und $\mathcal{L}_1$ ist eine echte Obermenge von $\mathcal{L}_2$.

Satz 5.23 Die Chomsky Hierarchie und die Platzkomplexität

(a) $\mathcal{L}_0$ *ist die Klasse aller rekursiv aufzählbaren Sprachen.*

(b) Es gilt $\mathcal{L}_1 = \mathtt{Nspace}(n)$*. Also ist* $\mathcal{L}_1$ *die Klasse aller Sprachen, die von nichtdeterministischen Turingmaschinen auf linearem Platz erkannt werden. Insbesondere ist jede Sprache in* $\mathcal{L}_1$ *entscheidbar.*

(c) $\mathtt{NL} \subseteq \mathtt{LOGCFL} \subseteq \mathtt{Dspace}(\log_2^2 n)$.
(Insbesondere sind alle kontextfreie Sprachen in $\mathtt{Dspace}(\log_2^2 n)$ *enthalten, es gilt also* $\mathcal{L}_2 \subseteq \mathtt{Dspace}(\log_2^2 n)$.*)*

(d) Die Klasse der regulären Sprachen stimmt mit der Klasse $\mathtt{Dspace}(0)$ *überein, es gilt also* $\mathcal{L}_3 = \mathtt{Dspace}(0)$.

(e) $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$ *und alle Inklusionen sind echt.*

Beweis (a): Wir behaupten, dass $L(G)$ für jede Grammatik G rekursiv aufzählbar ist. Warum? Wenn wir entscheiden wollen, ob ein Wort w zur Sprache $L(G)$ gehört, produzieren wir alle mögliche Ableitungen: Wenn $w \in L(G)$ werden wir eine Ableitung finden. Wenn $w \notin L(G)$, wird unser Programm nicht halten, aber dies ist auch nicht erforderlich.

Andererseits sei die Sprache L rekursiv aufzählbar. Es gibt also eine Turingmaschine M mit $L = L(M)$. Wir müssen eine Grammatik G mit $L(M) = L(G)$ konstruieren. Für die Konstruktion von G beachten wir, dass Berechnungen von M natürlich stets mit der Eingabe w beginnen, während eine Ableitung von w mit w endet. Also sollten wir die Grammatik so konstruieren, dass die Berechnungen von M „rückwärts" simuliert werden.

Zuerst normieren wir M. Wir nehmen an, dass M

- nur einen akzeptierenden Zustand q_a besitzt und
- dass akzeptierende Berechnungen mit dem leeren Band enden.

Wenn wir eine Berechnung von M anhalten, dann können wir die gegenwärtige Situation durch den

- Bandinhalt $\alpha_1 \cdots \alpha_N \in \Gamma^N$,
- den gegenwärtigen Zustand $q \in Q$ und
- die Position des Kopfes

exakt beschreiben. Wenn der Kopf die Position i des Bands liest, dann beschreiben wir die Konfiguration durch das Wort

$$\alpha_1 \cdots \alpha_{i-1}\, q\, \alpha_i \cdots \alpha_N.$$

Die zu entwerfende Grammatik G wird Konfigurationen von M rückwärts konstruieren. G besitzt

$$(\Gamma \setminus \Sigma) \cup Q \cup \{\varepsilon\}$$

als Variablenmenge und q_a als Startsymbol. Zuerst wird der von M benutzte Bandbereich durch die Produktionen $q_a \to \mathtt{B}q_a \mid q_a\mathtt{B}$ erzeugt. Dann beginnt die Rückwärtsrechnung.

Fall 1: $\delta(q, a) = (q', b, \text{links})$. Wir nehmen die Produktion

$$q'cb \to cqa \quad \text{für alle } c \in \Gamma$$

auf: Wenn die Konfiguration $* \cdots * q'cb * \cdots *$ schon erzeugt wurde, können wir damit die mögliche Vorgänger-Konfiguration $* \cdots * \, cqa \, * \cdots *$ erzeugen.

Fall 2: $\delta(q,a) = (q', b, \text{bleib})$. Wir fügen die Produktion

$$q'b \to qa.$$

zu G hinzu.

Fall 3: $\delta(q,a) = (q', b, \text{rechts})$. Diesmal nehmen wir die Produktion

$$bq' \to qa.$$

auf.

Am Ende der Ableitung werden wir eine Konfiguration

$$\mathtt{B}^k q_0 w \mathtt{B}^s$$

erzeugt haben. Die zusätzlichen Produktionen

$$\begin{array}{rcll} q_0 & \to & \varepsilon_1 & \\ \mathtt{B}\,\varepsilon_1 & \to & \varepsilon_1 & \\ \varepsilon_1 & \to & \varepsilon_2 & \\ \varepsilon_2 a & \to & a\varepsilon_2 & \text{für } a \in \Sigma \\ \varepsilon_2 & \to & \varepsilon_3 & \\ \varepsilon_3\,\mathtt{B} & \to & \varepsilon_3 & \\ \varepsilon_3 & \to & \text{das leere Wort} & \end{array}$$

garantieren jetzt, dass das Wort w abgeleitet wird und dass die Ableitung die Form $q_a \xrightarrow{*} \mathtt{B}^k q_0 w \mathtt{B}^s \xrightarrow{*} w$ hat. Insbesondere ist w genau dann ableitbar, wenn M die Konfigurationenfolge

$$\mathtt{B}^k q_0 w \mathtt{B}^s \xrightarrow{*} q_a$$

durchläuft.

(b) Zuerst beachten wir, dass eine kontextsensitive Grammatik längenerhaltend ist: Die rechte Seite v einer kontextsensitiven Produktion $u \to v$ ist mindestens so lang wie die linke Seite u. Wenn wir also aus dem Startsymbol ein Wort $w \in \Sigma^*$ erzeugen, dann sind alle zwischenzeitlich erzeugten Strings aus $(\Sigma \cup \Gamma)^*$ in ihrer Länge durch $|w|$ nach oben beschränkt. Das aber bedeutet, dass wir eine mögliche Ableitungsfolge auf Platz $O(|w|)$ raten und verifizieren können: Jede kontextsensitive Sprache kann also durch eine nichtdeterministische Turingmaschine auf linearem Platz erkannt werden.

Betrachten wir jetzt eine nichtdeterministische Turingmaschine M, die auf linearem Platz rechnet. Wir können zuerst annehmen, dass M sogar in-place arbeitet.

In Teil (a) haben wir eine beliebige Turingmaschine M durch Typ-0 Grammatiken simuliert und insbesondere die Äquivalenz

$$q_a \xrightarrow{*} \mathtt{B}^k q_0 w \mathtt{B}^s \;\Leftrightarrow\; w \in L(M)$$

erhalten. Sämtliche Produktionen der Ableitung $q_a \xrightarrow{*} \mathtt{B}^k q_0 w \mathtt{B}^s$ sind längenerhaltend. Wenn die Maschine M aber in-place arbeitet, erhalten wir deshalb die Äquivalenz

$$q_a \xrightarrow{*} q_0 w \;\Leftrightarrow\; w \in L(M).$$

Mit anderen Worten, wenn $L \in \texttt{Nspace}(n)$, dann besitzt q_0L eine kontextsensitive Grammatik. Wir sind fertig, denn:

Aufgabe 93
Wenn q_0L kontextsensitiv ist, dann ist auch L kontextsensitiv.

(c) Für die Beziehung

$$\texttt{NL} \subseteq \texttt{LOGCFL}$$

genügt der Nachweis, dass D-REACHABILITY mit einer LOGSPACE-Reduktion auf eine kontextfreie Sprache L reduziert werden kann. Wir beschreiben L, indem wir einen Kellerautomaten K angeben, der L akzeptiert.

K nimmt an, dass die Eingabe w ein Element von $(a^*b^*)^*$ ist und interpretiert ein Teilwort a^rb^s als die Kante von Knoten r nach Knoten s; ein mehrmaliges Auftreten von Kanten ist erlaubt.

K rät einen Weg von Knoten 1 nach Knoten 2,

- indem es eine erste Kante $(1, u)$ rät und u auf den Keller legt.
- Der Knoten v liege gegenwärtig auf dem Keller. K rät eine Kante (v', w), die in der Eingabe nach den bisher geratenen Kanten erscheint.
  - Mit Hilfe des Kellers verifiziert K, dass $v = v'$ gilt. Gilt $v \neq v'$, verwirft K.
  - K legt w auf den Keller und akzeptiert, wenn $w = 2$.
  - Für $w \neq 2$ wiederholt K sein Vorgehen.

Wir beschreiben eine LOGSPACE-Reduktion von D-REACHABILITY auf L. Für einen gerichteten Graphen G zählen wir die Anzahl n der Kanten. Danach geben wir die Kanten, in einer jeweils beliebigen Reihenfolge, genau $n - 1$ Mal aus. Der Kellerautomat K, wenn auf die Ausgabe angesetzt, findet genau dann einen Weg von Knoten 1 nach Knoten 2, wenn ein solcher Weg in G existiert.

Die verbleibende Beziehung „LOGCFL$\subseteq \texttt{Dspace}(\log_2^2 n)$“ folgt aus der nächsten Übungsaufgabe.

Aufgabe 94
Zeige, dass das Wortproblem für kontextfreie Sprachen in $\texttt{Dspace}(\log_2^2 n)$ liegt.

(d) Die Behauptung folgt aus Satz 5.6.

(e) Beachte, dass $\mathcal{L}_1$ eine echte Teilmenge von $\mathcal{L}_0$ ist, da alle Sprachen in $\mathcal{L}_1$ entscheidbar sind. Die restlichen (echten) Inklusionen haben wir in den obigen Beispielen nachgewiesen. □

Wie schwierig ist das Wortproblemn für die Klasse der kontextsensitiven Sprachen? Viel zu schwierig, denn das NP-vollständige Erfüllbarkeitsproblem ist kontextsensitiv. Aber die Situation ist sogar noch viel schlimmer, denn das Wortproblem ist, wie wir gleich sehen werden, sogar PSPACE-vollständig.

Satz 5.24 *(Das Wortproblem für kontextsensitive Sprachen)*

(a) Die Sprache KNF-SAT ist kontextsensitiv.

(b) Das Wortproblem für kontextsensitive Sprachen ist PSPACE*-vollständig.*

Beweis (a): Wir können natürlich eine erfüllende Belegung auf linearem Platz raten und verifizieren. Die Behauptung folgt also aus Satz 5.23 (b).

(b) ist eine Konsequenz der folgenden Übungsaufgabe. □

Aufgabe 95
Zeige, dass das Wortproblem für kontextsensitive Sprachen PSPACE-vollständig ist.

Wie verhalten sich die Klassen P, NP und PSPACE zu den Klassen der Chomsky-Hierarchie?

Satz 5.25

(a) $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathtt{P}$ *und alle Inklusionen sind echt.*

(b) $\mathcal{L}_1 \subset \mathtt{PSPACE} \subset \mathcal{L}_0$ *und alle Inklusionen sind echt.*

Beweis (a): Da das Wortproblem in Zeit $O(|w|^3)$ lösbar ist, gehören alle kontextfreien Sprachen zur Klasse P. Die Sprache $\{a^k b^k c^k \mid k \in \mathbb{N}\}$ ist nicht kontextfrei, gehört aber natürlich zur Klasse P. Also ist $\mathcal{L}_2$ eine echte Teilmenge von P.

(b) In Satz 5.23 haben wir gezeigt, dass $\mathcal{L}_1 = \mathtt{Nspace}(n)$ gilt. Damit folgt $\mathcal{L}_1 \subseteq \mathtt{Dspace}(n^2)$ aus dem Satz von Savitch. Schließlich können wir $\mathtt{Dspace}(n^2) \subset \mathtt{PSPACE}$ aus der Speicherplatz-Hierarchie von Satz 5.5 folgern.

Wiederum aus Satz 5.23 wissen wir, dass $\mathcal{L}_0$ mit der Klasse der rekursiv aufzählbaren Sprachen übereinstimmt. Da jede Sprache in PSPACE entscheidbar ist, muss die Inklusion PSPACE $\subset \mathcal{L}_0$ echt sein. □

5.6 Probabilistische Turingmaschinen und Quantenrechner

Um „wieviel mächtiger" sind probabilistische Turingmaschinen oder Quanten-Turingmaschinen im Vergleich zu „stink-normalen" deterministischen Turingmaschinen?

Wir führen zuerst probabilistische Turingmaschinen ein. Eine probabilistische Turingmaschine M wird durch den Vektor

$$M = (Q, \Sigma, \delta, q_0, \Gamma, F)$$

beschrieben. Die Überführungsfunktion δ hat, im Gegensatz zu deterministischen Turingmaschinen die Form

$$\delta : \Gamma \times Q \times \Gamma \times Q \times \{\mathtt{links}, \mathtt{bleib}, \mathtt{rechts}\} \longrightarrow [0,1] \cap \mathbb{Q}$$

und weist jedem möglichen Übergang

$$(\gamma, q) \longrightarrow (\gamma', q', \mathit{Richtung})$$

die Wahrscheinlichkeit

$$\delta(\gamma, q, \gamma', q', \mathit{Richtung})$$

zu. Wir verlangen, dass für jedes Paar $(\gamma, q) \in \Gamma \times Q$ eine Wahrscheinlichkeitsverteilung auf den Übergängen vorliegt. Das heißt, wir fordern für jedes Paar (γ, q) die Bedingung

$$\sum_{(\gamma', q', \mathit{Richtung}) \in \Gamma \times Q \times \{\mathtt{links}, \mathtt{bleib}, \mathtt{rechts}\}} \delta(\gamma, q, \gamma', q', \mathit{Richtung}) = 1.$$

Wie arbeitet eine probabilistische Turingmaschine M? Für Eingabe x wird M potenziell viele Berechnungen ausführen. Als Berechnung bezeichnen wir dabei die Folge von Konfigurationen

$$B \;:\; C_0 \to C_1 \to C_2 \to \ldots \to C_m,$$

die die Maschine durchläuft. Die Konfiguration C_0 ist die Konfiguration, bei der der Bandinhalt die Eingabe ist, der aktuelle Zustand der Anfangszustand q_0 und der Lese-/Schreibkopf in der Ausgangsposition ist. Als Wahrscheinlichkeit einer Berechnung bezeichnen wir das Produkt aller Übergangswahrscheinlichkeiten. Also

$$prob(B) = \prod_{i=0}^{m-1} p_i$$

wenn p_i die Wahrscheinlichkeit des Übergangs von Konfiguration C_i zu Konfiguration C_{i+1} bezeichnet. Wie sollen wir die von einer probabilistischen Turingmaschine M akzeptierte Sprache definieren? Es liegt nahe, für jede Eingabe x die Wahrscheinlichkeit akzeptierender Berechnungen, also

$$p_x = \sum_{B \text{ ist akzeptierende Berechnung von } x} prob(B)$$

zu messen.

Definition 5.26 *Sei M eine probabilistische Turingmaschine, dann ist*

$$L_M = \left\{ x \in \Sigma^* | p_x > \frac{1}{2} \right\}$$

die von M akzeptierte Sprache. Wir sagen, dass M beschränkten Fehler *besitzt, falls es $\varepsilon > 0$ gibt, so dass stets $p_x \leq \frac{1}{2} - \varepsilon$ gilt, falls x* nicht *zur Sprache L gehört und $p_x \geq \frac{1}{2} + \varepsilon$ gilt, falls x zur Sprache L gehört.*

Probabilistische Berechnungen mit beschränktem Fehler sind ein sinnvolles und in der Praxis nützliches Berechnungsmodell: Wenn wir eine Berechnung k-mal für eine vorgegebene Eingabe x laufen lassen und das Mehrheitsergebnis übernehmen, dann wird die Wahrscheinlichkeit eines Fehlers höchstens

$$2^{-\Omega(k)}$$

sein. (Warum ?)

Die Berechnungskraft probabilistischer Turingmaschinen mit unbeschränktem Fehler hingegen ist immens. So kann man zum Beispiel zeigen, dass nichtdeterministische Turingmaschinen ohne Zeitverlust simuliert werden können.

Satz 5.27 *Sei M eine probabilistische Turingmaschine (mit nicht notwendigerweise beschränktem Fehler). Wenn die worst-case Laufzeit einer jeden Berechnung für Eingaben der Länge n durch $t(n)$ beschränkt ist, dann gilt*

$$L(M) \in \texttt{Dspace}(t).$$

Beweisskizze : Die höchstens $2^{O(t(n))}$ Berechnungen für eine vorgegebene Eingabe x werden nacheinander simuliert. Ein Zähler summiert die Wahrscheinlichkeiten akzeptierender Berechnungen (auf $O(t(n))$ Zellen). Nachdem alle Berechnungen simuliert sind, wird geprüft, ob der Zähler einen Wert größer $\frac{1}{2}$ hat, und in diesem Fall wird akzeptiert. $\square$

Wir kommen als nächstes zu einer allerdings nur recht oberflächlichen Beschreibung von Quantenrechnern. Zu Anfang erinnern wir an das Rechnen mit komplexen Zahlen. $\mathbb{C} = \{x+iy \mid x, y \in \mathbb{R}\}$ bezeichnet die Menge der komplexen Zahlen und es ist $i = \sqrt{-1}$. Für die komplexe Zahl $z = x + iy$ ist $\bar{z} = x - iy$ die Konjugierte von z. Die Länge von z ist durch

$$|z| = \sqrt{x^2 + y^2}$$

definiert und für komplexe Zahlen $z_1, z_2 \in \mathbb{C}$ mit $z_k = x_k + iy_k$ ist

$$\begin{aligned} z_1 + z_2 &= x_1 + x_2 + i(y_1 + y_2) \\ z_1 \cdot z_2 &= x_1 \cdot x_2 - y_1 \cdot y_2 + i(x_1 \cdot y_2 + x_2 \cdot y_1). \end{aligned}$$

Die Grobstruktur eines Quantenrechners ähnelt der einer probabilistischer Turingmaschine. Diesmal hat aber die Überführungsfunktion δ die Form

$$\delta \ : \ \Gamma \times Q \times \Gamma \times Q \times \{\texttt{links}, \texttt{bleib}, \texttt{rechts}\} \longrightarrow \mathbb{Q} + i\mathbb{Q}$$

wobei nur komplexe Zahlen der Länge höchstens 1 zugewiesen werden. Wie im Fall probabilistischer Turingmaschinen gibt es zu jedem Paar $(\gamma, q) \in \Gamma \times Q$ potentiell viele Übergänge, wobei diesmal

$$\sum_{\gamma', q', Richtung} |\delta(\gamma, q, \gamma', q', Richtung|^2 = 1$$

gelten muß. Wir sagen, dass

$$\delta(\gamma, q, \gamma', q', Richtung)$$

die (Wahrscheinlichkeits-)Amplitude ist und, dass

$$|\delta(\gamma, q, \gamma', q', Richtung)|^2$$

die zugewiesene Wahrscheinlichkeit ist. Bisher haben wir nur eine merkwürdige Darstellung der Wahrscheinlichkeit eines Übergangs kennengelernt, der wesentliche Unterschied zu den probabilistischen Turingmaschinen folgt aber sofort: Wir weisen jeder Berechnung B das Produkt p_B der ihren Übergängen entsprechenden Wahrscheinlichkeitsamplituden zu. Charakteristischerweise werden wir im Allgemeinen aber viele Berechnungen haben, die in derselben Konfiguration C enden. Wir weisen der Konfiguration C die Wahrscheinlichkeitsamplitude

$$\tau_C = \sum_{B \text{ führt auf } C} p_B$$

zu und definieren

$$|\tau_C|^2$$

als die Wahrscheinlichkeit der Konfiguration C. Die von einem Quantenrechner Q akzeptierte Sprache definieren wir dann als

$$L(Q) = \left\{ x | \sum_{C \text{ ist akzeptierende Konfiguration von } Q \text{ auf Eingabe } x} |\tau_C|^2 > \frac{1}{2}, \right\}$$

analog zu probabilistischen Turingmaschinen.

Unsere Beschreibung ist zu diesem Zeitpunkt unvollständig: Das beschriebene Rechnermodell ist weitaus mächtiger als das Modell der Quantenrechner. Deshalb noch einmal ein Ausflug in die komplexe Zahlen.

Für eine Matrix $A = (z_{i,j})_{1 \leq i,j \leq n}$ mit komplexwertigen Einträgen ist

$$\overline{A} = (\overline{z_{j,i}})_{1 \leq i,j \leq n}$$

die konjugiert Transponierte von A. Wir nennen A *unitär*, falls

$$\overline{A} \cdot A = \text{Einheitsmatrix}.$$

Wir halten jetzt in der Konfigurationsmatrix A_Q die Wahrscheinlichkeitsamplituden eines 1-Schritt Übergangs zwischen je zwei Konfigurationen C und C' fest. Also

$$A_Q[C, C'] = \text{Wahrscheinlichkeitsamplitude des Übergangs von } C \text{ nach } C'.$$

Eine Quantenberechnung liegt vor, wenn die Matrix A_Q unitär ist.

Satz 5.28 *Wenn ein Quantenrechner Q die Sprache $L(Q)$ in Zeit $t(n)$ akzeptiert, dann ist*

$$L(Q) \in \texttt{Dspace}(t^2).$$

Hierzu ist die Forderung eines beschränkten Fehlers ebenso nicht notwendig wie die Forderung, dass die Konfigurationsmatrix A_Q unitär ist.

Beweisskizze : Die simulierende deterministische Turingmaschine wird die Einträge des Matrix/Vektor-Produkts

$$A_Q^{t(n)} \cdot v$$

nacheinander berechnen und die Wahrscheinlichkeiten akzeptierender Konfigurationen aufsummieren. Es wird akzeptiert, falls die Summe größer als 1/2 ist. Warum funktioniert dieser Ansatz, wenn wir den Vektor v durch

$$v_i = \begin{cases} 0 & i \neq \text{Startkonfiguration} \\ 1 & \textit{sonst} \end{cases}$$

definieren? Der Vektor $A_Q \cdot v$ gibt die Wahrscheinlichkeitsamplituden der 1-Schritt Nachfolger der Startkonfiguration wieder und allgemeiner listet der Vektor $A_Q^k \cdot v$ die Wahrscheinlichkeitsamplituden der k-Schritt Nachfolger auf. Im letzten Schritt (also $k = t(n)$) müssen wir dann nur noch von den Wahrscheinlichkeitsamplituden zu den Wahrscheinlichkeiten übergehen.

Wie berechnet man aber $A_Q^{t(n)} \cdot v$ in Platz $O(t^2(n))$? Die Matrix A_Q besitzt ja $2^{O(t(n))}$ Zeilen und Spalten! Hier ist ein Tip: Der Vektor $A_Q^k \cdot v$ kann in Platz $O(k \cdot t(n))$ berechnet werden. □

Selbst bei unbeschränktem Fehler liegen also Sprachen, die von probabilistischen Turingmaschinen oder von Quanten-Turingmaschinen in polynomieller Zeit akzeptiert werden, in `PSPACE`. Diese Aussage gilt selbst bei unbeschränktem Fehler.

5.7 Zusammenfassung

Wir haben zuerst deterministische Speicherplatzklassen untersucht. Wir haben gesehen, dass die auf Speicherplatz $o(\log_2 \log_2 n)$ berechenbaren Sprachen mit den regulären Sprachen übereinstimmen und deshalb „bringt ein zu kleiner Speicherplatz nichts Neues“.

Die erste nicht-triviale Klasse ist die Klasse DL aller auf logarithmischem Platz berechenbaren Sprachen. Wir haben die Grenzen der Berechnungskraft von DL am Beispiel von D-REACHABILITY betrachtet. Wir haben das Konzept der Log-Space Reduktionen entwickelt, um zu zeigen, dass D-REACHABILITY NL-vollständig ist, also ein schwierigstes Problem in NL ist, der Klasse aller in logarithmischem Platz nichtdeterministisch lösbaren Entscheidungsprobleme. (Insbesondere folgt, dass NL in P enthalten ist.) Weitere NL-vollständige Probleme sind 2-SAT, das Wortproblem für NFA und der Test auf Bipartitness.

Der Satz von Savitch weist nach, dass Nichtdeterminismus nur zu einem quadratischen Speicherplatzgewinn führt, da $\texttt{Nspace}(s) \subseteq \texttt{Dspace}(s^2)$ für platz-konstruierbare Funktionen s gilt. Auch das Komplementverhalten ist „nicht typisch für Nichtdeterminismus", denn es ist überraschenderweise $\texttt{Nspace}(s) = \text{co}\texttt{Nspace}(s)$, falls s platz-konstruierbar ist.

Schließlich haben wir die wichtige Komplexitätsklasse PSPACE definiert. PSPACE lässt sich als die Komplexitätsklasse nicht-trivialer Zwei-Personen Spiele auffassen, da das Problem der quantifizierten Booleschen Formeln PSPACE-vollständig ist. Entscheidungsprobleme für reguläre Ausdrücke oder NFA, wie die Universalität, das Äquivalenzproblem oder die Minimierung, haben sich als unanständig schwierig, nämlich als PSPACE-hart herausgestellt.

Die Klasse PSPACE ist mächtig und enthält alle Entscheidungsprobleme, die durch randomisierte Algorithmen oder Quanten-Algorithmen in polynomieller Zeit lösbar sind.

Wir haben dann die Chomsky-Hierarchie betrachtet. Die rekursiv aufzählbaren Sprachen sind genau die Sprachen, die von unbeschränkten Grammatiken erzeugt werden können. Selbst die kontextsensitiven Sprachen sind noch zu komplex, da ihr Wortproblem PSPACE-vollständig sein kann. (Beachte, dass die Klasse PSPACE die Klasse NP enthält und dementsprechend ist eine Lösung eines PSPACE-vollständigen Wortproblems in aller Wahrscheinlichkeit noch sehr viel komplexer als die Lösung eines NP-vollständigen Problems.) Demgegenüber stehen die kontextfreien Sprachen, deren Wortproblem bei Beschränkung auf deterministisch kontextfreie Sprachen sogar in Linearzeit gelöst werden kann.

Kapitel 6

Parallelität

Das Textbuch „Limits to Parallel Computation: P-Completeness Theory“, von Raymond Greenlaw, James Hoover und Walter Ruzzo, Oxford University Press (1995) ist eine sehr gute Referenz für die Inhalte dieses Kapitels[1].

Wann ist ein algorithmisches Problem parallelisierbar? Was soll „parallelisierbar“ überhaupt bedeuten? Informell gesprochen sollten wir fordern, dass Berechnungen rasant schnell ablaufen *und* mit nicht zu vielen Prozessoren arbeiten. Um diese Begriffe weiter zu klären, wählen wir Schaltkreise als unser paralleles Rechnermodell.

Definition 6.1 *Ein Schaltkreis S wird durch den Vektor*

$$S = (G, Q, R, gatter, n, eingabe)$$

beschrieben.

- *$G = (V, E)$ ist ein gerichteter, azyklischer Graph.*
- *Q ist die Menge der Quellen (Knoten mit nur ausgehenden Kanten) und R die Menge der Senken (Knoten mit nur eingehenden Kanten) von G.*
- *Die Funktion eingabe: $Q \to \{1, \ldots, n\}$ weist jeder Quelle die Position des entsprechenden Eingabebits zu.*
- *Die Funktion gatter: $V \setminus Q \to \{\neg, \vee, \wedge\}$ weist jedem inneren Knoten von G eine Boolesche Funktion zu, wobei einem Knoten v die Negation nur zugewiesen werden darf, wenn v genau eine eingehende Kante besitzt.*

Der Schaltkreis berechnet die Boolesche Funktion $f_S : \{0,1\}^n \to \{0,1\}^{|R|}$, indem die n Eingabebits an die Quellen in G angelegt werden, und jeder Knoten das Ergebnis seiner Gatterfunktion weiterleitet. Das Resultat wird an den Senken von G (in einer vorher fixierten Reihenfolge) abgelesen.

Als nächstes führen wir die Komplexitätsmaße Tiefe, Größe und Fanin ein. Während die Tiefe die parallele Rechenzeit wiedergibt, misst die Größe die Prozessoranzahl, bzw. in gewissem Sinne die sequentielle Rechenzeit.

[1] Das Buch kann auch über die Webseite `http://www.cs.armstrong.edu/greenlaw/research/PARALLEL/limits.pdf` heruntergeladen werden.

Definition 6.2 *Sei $S = (G, Q, R, gatter, n, eingabe)$ ein Schaltkreis. Wir sagen, dass*

(a) die Tiefe von S die Länge des längsten Weges in G ist,

(b) die Größe von S die Anzahl der Knoten von G ist, wobei wir keine Quellen zählen,

(c) und der Fanin von S das Maximum, über alle Knoten v, der Anzahl eingehender Kanten von v ist.

Aufgabe 96
Zeige, dass das Problem, die Summe von n Bits modulo 2 zu bestimmen, nicht von Schaltkreisen mit unbeschränktem Fanin und Tiefe 2 gelöst werden kann, wenn polynomielle Größe in n gefordert wird.

Aufgabe 97
a) Bestimme die Anzahl der Funktionen $f : \{0,1\}^n \to \{0,1\}$.

b) Gib eine möglichst gute obere Schranke für die Anzahl der Schaltkreise aus m Gattern mit Fanin 2 und n Eingaben an.

c) Zeige: Es gibt eine Funktion $f : \{0,1\}^n \to \{0,1\}$, zu deren Berechnung Schaltkreise die Größe mindestens $\Omega(2^n/n)$ benötigen.

Schaltkreise berechnen nur Funktionen mit einer festen Anzahl von Eingabebits. Um Funktionen von unbeschränkt vielen Eingabebits berechnen zu können, benötigen wir das Konzept einer uniformen Schaltkreisfamilie.

Definition 6.3 *(a) Eine Schaltkreisfamilie ist eine Folge $(S_n)_{n \in \mathbb{N}}$ von Schaltkreisen, so dass S_n eine Boolesche Funktion auf genau n Eingaben berechnet. Eine Schaltkreisfamilie $(S_n)_{n \in \mathbb{N}}$ berechnet eine Funktion $f : \{0,1\}^* \to \{0,1\}$ genau dann, wenn S_n für jedes n die Funktion f, eingeschränkt auf $\{0,1\}^n$, berechnet.*

(b) Eine Schaltkreisfamilie $(S_n)_{n \in \mathbb{N}}$ ist uniform, wenn es eine $\log_2(\text{Größe}(S_n) + n)$-platzbeschränkte, deterministische Turingmaschine gibt, die für Eingabe 1^n

- *alle Knoten von S_n aufzählt,*
- *jedem inneren Knoten seine Funktion, also eine Funktion aus $\{\neg, \vee, \wedge\}$, zuweist,*
- *sämtliche Kanten von S_n aufzählt und*
- *jeder Quelle von S_n eine Bitposition zuweist.*

(c) Seien $d, s : \mathbb{N} \to \mathbb{N}$ gegeben. Definiere

$$\texttt{DEPTH}_{\text{uniform}}(d)$$

als die Klasse aller Sprachen L, die durch eine uniforme Schaltkreisfamilie in Tiefe $O(d(n))$ und mit Fanin zwei berechnet werden.

$$\texttt{SIZE}_{\text{uniform}}(s)$$

ist die Klasse aller Sprachen L, die durch eine uniforme Schaltkreisfamilie mit Größe $O(s(n))$ und mit Fanin zwei berechnet werden. Schließlich besteht

$$\texttt{DEPTH}{-}\texttt{SIZE}_{\text{uniform}}(d, s) = \texttt{DEPTH}_{\text{uniform}}(d) \cap \texttt{SIZE}_{\text{uniform}}(s)$$

aus allen Sprachen, die sowohl in Tiefe $O(d)$ wie auch in Größe $O(s)$ durch eine uniforme Schaltkreisfamilie vom Fanin zwei berechnet werden.

Mit anderen Worten: Wir verlangen, dass eine uniforme Schaltkreisfamilie S_n einfach, nämlich durch eine logarithmisch-bandbeschränkte Turingmaschine, konstruierbar sein muss. Warum haben wir die Uniformität oder Einfachheit von Schaltkreisen gefordert?

Sei n die Gödelnummer der Turingmaschine M. Wenn M auf dem leeren Wort hält, dann definieren wir S_n so, dass alle Eingaben akzeptiert werden. Wenn hingegen M auf dem leeren Wort nicht hält, dann definiere S_n so, dass alle Eingaben verworfen werden. Mit anderen Worten, das spezielle Halteproblem (für die Unärkodierung von Gödelnummern) kann mit einer Schaltkreisfamilie berechnet werden, die nur aus Schaltkreisen mit einem Gatter bestehen. Die beschriebene Schaltkreisfamilie $(S_n)_{n \in \mathbb{N}}$ ist nicht uniform und ihre Konstruktion ist sogar nicht berechenbar.

In uniformen Schaltkreisfamilien wird die wesentliche Arbeit von den Schaltkreisen geleistet und nicht von der Beschreibung der Familie.

Aufgabe 98
a) Zeige: Für jede Funktion $f : \{0,1\}^* \to \{0,1\}$ gibt es eine Schaltkreisfamilie der Tiefe $O(n)$, die f berechnet.

b) Zeige: Es gibt eine berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}$, die nicht von einer uniformen Schaltkreisfamilie polynomieller Tiefe berechnet werden kann.

Wir sind an rasant schnellen, parallelen Algorithmen interessiert, die mit einer vernünftigen Größe auskommen. Die Begriffe „rasant schnell“ und „vernünftige Größe“ präzisieren wir durch poly-logarithmische Rechenzeit und polynomielle Größe.

Definition 6.4 *Für jedes $k \in \mathbb{N}$ definiere die Komplexitätsklassen:*

$$\begin{aligned} \mathtt{NC}^k &:= \bigcup_{l=0}^{\infty} \mathtt{DEPTH}-\mathtt{SIZE}_{\text{uniform}}(\log_2^k n, n^l) \\ \mathtt{NC} &:= \bigcup_{k \in \mathbb{N}} \mathtt{NC}^k \\ \mathtt{AC}^k & \quad \textit{ist wie } \mathtt{NC}^k \textit{ definiert, allerdings beschränken wir den Fanin nicht.} \\ \mathtt{AC} &:= \bigcup_{k \in \mathbb{N}} \mathtt{AC}^k. \end{aligned}$$

Die Komplexitätsklasse $\mathtt{NC}$ ist unsere Definition der Klasse parallelisierbarer Probleme. $\mathtt{NC}$ steht für „Nick's Class“ (nach ihrem Autor Nick Pippenger). $\mathtt{AC}$ steht für „Alternating Circuits“, d.h. Schaltkreise die alternierend aus Schichten von UND- bzw. ODER-Gattern aufgebaut sind.

Der folgende Satz stellt eine Beziehungen zwischen den eingeführten Komplexitätsklassen und zu $\mathtt{P}$ her.

Satz 6.5 *Für alle $k \in \mathbb{N}$ gilt*

(a) $\mathtt{AC}^k \subseteq \mathtt{NC}^{k+1} \subseteq \mathtt{AC}^{k+1}$.

(b) $\mathtt{AC} = \mathtt{NC} \subseteq \mathtt{P}$.

Beweis (a): Die Inklusion $\mathtt{NC}^{k+1} \subseteq \mathtt{AC}^{k+1}$ ist offensichtlich. Wir zeigen $\mathtt{AC}^k \subseteq \mathtt{NC}^{k+1}$. Sei $(S_n)_{n \in \mathbb{N}}$ eine uniforme Schaltkreisefamilie, die eine Sprache aus $\mathtt{AC}^k$ akzeptiert. Ein Und- bzw. Oder-Gatter mit p Eingängen kann durch einen binären Baum der Tiefe $\lceil \log_2 p \rceil$ und Größe höchstens $2p+1$ simuliert werden. Da der Fanin von S_n durch n + Größe(S_n) beschränkt ist und da Größe(S_n) polynomiell in der Eingabelänge n ist, führt die Ersetzung der Knoten von S_n durch

Binärbäume auf eine um höchstens den Faktor $O(\log_2 n)$ größere Tiefe. Beachte, daß die Größe durch die Ersetzungen höchstens quadriert wird.

(b) Die Gleichheit $\mathtt{AC} = \mathtt{NC}$ folgt aus Teil (a). Betrachten wir die Inklusion $\mathtt{NC} \subseteq \mathtt{P}$. Sei $L \in \mathtt{NC}$. Dann gibt es eine uniforme Schaltkreisfamilie $(S_n)_{n\in\mathbb{N}}$, die L berechnet. Es genügt zu zeigen, daß S_n in polynomieller Zeit konstruierbar und auswertbar ist. Die polynomielle Konstruierbarkeit folgt, da S_n durch eine logarithmisch-platzbeschränkte Turingmaschine berechenbar ist. Eine Auswertung gelingt mit Tiefensuche in polynomieller Zeit, da Größe$(S_n) = \text{poly}(n)$. □

Aufgabe 99
Zeige, dass man zwei Binärzahlen in $\mathtt{AC}^0$ addieren kann (d.h. jedes Bit der Summe in $\mathtt{AC}^0$ berechnen kann).

Aufgabe 100
Es sei ein nichtnegativ gewichteter, gerichteter Graph als Distanzmatrix mit Einträgen einer bestimmten Bitlänge gegeben. Es soll für alle Knotenpaare die Länge des kürzesten Weges zwischen ihren Knoten bestimmt werden. Hierzu wird ein paralleler Algorithmus gesucht, der ähnlich wie Floyds Algorithmus arbeitet. Der Algorithmus soll mit den Ressourcen von $\mathtt{NC}^2$ arbeiten.

*SC*Hinweis: Betrachte eine Matrixmultiplikation, bei der statt der normalen Multiplikation die Addition und statt der normalen Addition die Minimumbestimmung verwendet wird und wende diese auf das Problem an.

Aufgabe 101
Eine Formel ist ein Schaltkreis mit Fanin 2 und Fanout 1, d.h. die Graphstruktur einer Formel ist ein Baum. Die Größe einer Formel ist die Anzahl der Blätter.

(a) Zeige: Jede Funktion in $\mathtt{NC}^1$ hat eine polynomiell große Formel.

(b) Zeige: Wenn f eine Formel der Größe N hat, dann hat f eine äquivalente Formel der Tiefe $O(\log_2 N)$.

Aufgabe 102

Die Klasse $\mathtt{N\text{-}AC}^0_2$ bestehe aus uniformen Schaltkreisfamilien mit unbeschränktem Fanin und Tiefe 2, die nichtdeterministisch arbeiten, d.h., die zusätzlich zur „normalen“ Eingabe ein nichtdeterministisches Ratewort lesen und entsprechend akzeptieren.

Zeige: $\mathtt{NP} = \mathtt{N\text{-}AC}^0_2$.

6.1 Parallele Rechenzeit versus Speicherplatz

Welche Funktionen können uniforme Schaltkreisfamilien in Tiefe $s(n)$ berechnen? Angenommen, die uniforme Schaltkreisfamilie $(S_n)_{n\in\mathbb{N}}$ hat Tiefe $s(n) = \Omega(\log_2 n)$. Wir führen auf dem Schaltkreis in umgekehrter Richtung, also von der Senke zu den Quellen, eine Tiefensuche durch, um die Ausgabe des Schaltkreises S_n speicherplatz-effizient zu berechnen. Anstatt den gesamten Schaltkreis zu speichern, leiten wir jedes Mal die für den Abruf einer speziellen benötigten Information über S_n alle durch die Beschreibung des Schaltkreises gegebenen Informationen. Das gelingt mit Speicherplatz $O(s(n))$.

Wir speichern den Weg der Tiefensuche durch die Bitfolge $\vec{b}$ ab. Falls $b_i = 1$ (bzw. $b_i = 0$), ist der $(i+1)$-te Knoten des Wegs der rechte (bzw. linke) Nachfolger des i-ten Knoten. Die Länge der Liste ist proportional zur Tiefe $s(n)$ und die Turingmaschine rechnet mit Speicherplatz $O(s(n))$. Wir haben also gerade nachgewiesen, dass

$$\mathtt{DEPTH}_{\text{uniform}}(s) \subseteq \mathtt{Dspace}(s)$$

gilt. Um die umgekehrte Fragestellung, nämlich die Simulation von „Speicherplatz“ durch „Tiefe“ zu untersuchen, betrachten wir die transitive Hülle von Graphen.

Lemma 6.6 *Konstruiere uniforme Schaltkreisfamilien (mit unbeschränktem) Fanin, so dass*

(a) *zwei Boolesche $n \times n$-Matrizen in Tiefe zwei mit $O(n^3)$ Gattern multipliziert werden und*

(b) *die transitive Hülle eines Graphen in Tiefe $O(\log_2 n)$ und Größe $O(n^3 \log_2 n)$ mit unbeschränktem Fanin berechnet wird.*

Beweis (a): Offensichtlich kann ein Schaltkreis das Produkt

$$(A \cdot B)[i, j] = \bigvee_{k=1}^{n} A[i, k] \wedge B[k, j]$$

in Tiefe zwei mit $O(n^3)$ Gattern berechnen.

(b) Sei A die Adjazenzmatrix eines gerichteten Graphen mit n Knoten und sei E_n die $n \times n$ Einheitsmatrix. Man beweist durch Induktion über d, dass es genau dann einen Weg der Länge d von i nach j gibt, wenn $A^d[i, j] = 1$.

Ein Weg von i nach j darf auf jedem seiner Knoten für mehrere Schritte verweilen. Mit anderen Worten, es gibt genau dann einen Weg von i nach j, wenn $(A \vee E_n)^n[i, j] = 1$. Die Matrix $(A \vee E_n)^n$ kann mit wiederholtem Quadrieren für eine Zweierpotenz n schnell berechnet werden.

$B := A \vee E_n$;
FOR $i = 1$ TO $\log_2 n$ DO
    $B := B^2$;

Jeder Quadrierungsschritt gelingt in Tiefe zwei und Größe $O(n^3)$ ist ausreichend. Wir haben durch Übereinandersetzen von $\log_2 n$ Schaltkreisen zur Matrizenmultiplikation die transitive Hülle in Tiefe $O(\log_2 n)$ und Größe $O(n^3 \log_2 n)$ berechnet. Beachte, daß die erhaltene Schaltkreisfamilie uniform ist (Warum? Wie geht man vor, wenn n keine Zweierpotenz ist?). $\square$

Wir können jetzt die enge Kopplung zwischen den Komplexitätsmaßen „Speicherplatz“ und „Tiefe“ präzisieren.

Satz 6.7 *Sei $s : \mathbb{N} \to \mathbb{N}$ mit $s(n) = \Omega(\log_2 n)$ gegeben. Die Funktion s sei platz-konstruierbar. Dann ist*

$$\texttt{Dspace}(s) \subseteq \texttt{Nspace}(s) \subseteq \texttt{DEPTH} - \texttt{SIZE}_{\text{uniform}}(s^2, 2^{O(s)}) \subseteq \texttt{Dspace}(s^2).$$

Beweis: Wir wissen, dass $\texttt{DEPTH}_{\text{uniform}}(s) \subseteq \texttt{Dspace}(s)$ gilt. Also genügt der Nachweis von $\texttt{Nspace}(s) \subseteq \texttt{DEPTH} - \texttt{SIZE}_{\text{uniform}}(s^2, 2^{O(s)})$. Sei also M eine nichtdeterministische Turingmaschine, die mit Speicherplatz höchstens s rechnet. Für Eingabe w betrachten wir den Berechnungsgraphen $G_M(w)$. Wir haben zu entscheiden, ob es einen Weg vom Startknoten zu einem akzeptierenden Knoten in $G_M(w)$ gibt. Offensichtlich kann man annehmen, dass es genau eine akzeptierende Konfiguration gibt, und wir müssen das Problem D-REACHABILITY für $G_M(w)$ lösen

$G_M(w)$ hat höchstens $N = 2^{O(s(n))}$ Knoten. Wir berechnen zuerst $G_M(w)$ und lösen sodann D-REACHABILITY mit Lemma 6.6 in Tiefe $O(\log_2 N) = O(s)$ und Größe $O(s(n)N^3) = 2^{O(s(n))}$ durch eine uniforme Schaltkreisfamilie[2]. Um den Fanin von maximal $2^{O(s(n))}$ auf zwei zu drücken, muss die Tiefe von $O(s)$ auf $O(s^2)$ erhöht werden, die Größe ändert sich dabei höchstens polynomiell. $\square$

Aufgabe 103
Zeige, dass $\texttt{NL} \subseteq \texttt{AC}^1$ gilt.

---

[2]Um die Schaltkreisfamilie uniform zu konstruieren, muss die Platzkonstruierbarkeit von $s(n)$ gefordert werden.

6.2 $\mathtt{P}$-Vollständigkeit

Wir möchten die vom Standpunkt der Parallelisierbarkeit „schwierigsten“ Probleme in $\mathtt{P}$ bestimmen und wählen die LOGSPACE-Reduzierbarkeit, um die Parallelisierbarkeit zweier vorgegebener Sprachen zu vergleichen. Weshalb betrachten wir die LOGSPACE-Reduktion?

Lemma 6.8

(a) $\mathtt{DL} \subseteq \mathtt{NC}^2$.

(b) Zwei Sprachen $L_1, L_2 \subseteq \{0,1\}^*$ *seien gegeben. Aus* $L_1 \leq_{\text{LOG}} L_2$ *und* $L_2 \in \mathtt{NC}$ *folgt* $L_1 \in \mathtt{NC}$.

Beweis (a): ist eine direkte Konsequenz von Satz 6.7.

(b) Nach Definition existiert eine deterministische, logarithmisch-platzbeschränkte Turingmaschine M mit

$$w \in L_1 \Leftrightarrow M(w) \in L_2.$$

Da M logarithmisch-platzbeschränkt ist, kann jedes Ausgabebit von M durch eine Schaltkreisfamilie $(T_n | n \in \mathbb{N})$ mit polynomieller Größe und Tiefe $O(\log_2^2 n)$ berechnet werden, denn $\mathtt{DL} \subseteq \mathtt{NC}^2$. Wenn also L_2 durch die uniforme Schaltkreisfamilie $(S_n | n \in \mathbb{N})$ polynomieller Größe und polylogarithmischer Tiefe erkannt wird, so ist auch L_1 in polynomieller Größe und polylogarithmischer Tiefe erkennbar. Daher ist $L_1 \in \mathtt{NC}$. □

Da $\mathtt{DL} \subseteq \mathtt{NC}^2$ gilt, sind alle Sprachen in $\mathtt{DL}$ parallelisierbar. Insbesondere ist die LOGSPACE-Reduktion durch einen Schaltkreis polynomieller Größe und der Tiefe $O(\log^2 n)$ simulierbar. Aus Teil (b) folgt aus $L_1 \leq_{\text{LOG}} L_2$, dass L_1 parallelisierbar ist, wenn L_2 parallelisierbar ist. Trotzdem ist Lemma 6.8 kein entscheidender Grund für die LOGSPACE-Reduktion, da die Aussage auch für andere Reduktionen gilt. Zum Beispiel könnten wir statt LOGSPACE-Reduktionen als Transformation eine uniforme Schaltkreisfamilie mit Tiefe $\text{poly}(\log_2 n)$ und Größe $\text{poly}(n)$ zulassen. Zwar ist ein solcher Ansatz legal und wird gelegentlich auch benutzt, allerdings werden meistens „starke“ Transformationen nicht benötigt: Die „schwache“ LOGSPACE-Reduktion genügt sogar, um die wichtigsten $\mathtt{NP}$-Vollständigkeitsergebnisse zu etablieren. Wir arbeiten daher im weiteren nur mit der LOGSPACE-Reduktion.

Definition 6.9

(a) Eine Sprache L heißt genau dann $\mathtt{P}$-hart, wenn $K \leq_{\text{LOG}} L$ für alle Sprachen $K \in \mathtt{P}$ gilt.

(b) Eine Sprache L heißt genau dann $\mathtt{P}$-vollständig, wenn $L \in \mathtt{P}$ und wenn L $\mathtt{P}$-hart ist.

Die $\mathtt{P}$-vollständigen Sprachen sind die vom Standpunkt der Parallelisierbarkeit die schwierigsten Sprachen in $\mathtt{P}$: Wenn eine $\mathtt{P}$-vollständige Sprache parallelisierbar ist, dann stimmen $\mathtt{NC}$ und $\mathtt{P}$ überein und alle Sprachen in $\mathtt{P}$ wären überraschenderweise parallelisierbar.

Lemma 6.10 *Sei L eine $\mathtt{P}$-vollständige Sprache. Dann gilt:*

$$\mathtt{P} = \mathtt{NC} \Leftrightarrow L \in \mathtt{NC}.$$

Beweis $\Rightarrow$: Da $L \in \mathtt{P}$ und da nach Voraussetzung $\mathtt{P} = \mathtt{NC}$, ist $L \in \mathtt{NC}$.

$\Leftarrow$. Nach Voraussetzung ist $L \in \mathtt{NC}$. Da L $\mathtt{P}$-vollständig ist, gilt $K \leq_{\text{LOG}} L$ für alle $K \in \mathtt{P}$. Aus Lemma 6.8 folgt $K \in \mathtt{NC}$ für alle $K \in \mathtt{P}$, also folgt $\mathtt{P} \subseteq \mathtt{NC}$. Nach Satz 6.5 gilt $\mathtt{NC} \subseteq \mathtt{P}$, und wir erhalten die Behauptung. □

6.2.1 Das Circuit Value Problem

Für einen Schaltkreis S sei $\langle S\rangle$ die Binärkodierung eines Programmes, das

- die Gatter von S in irgendeiner Reihenfolge aufzählt,
- jedem Gatter seine Funktion zuweist,
- sämtliche Kanten aufzählt und
- jeder Quelle eine Bitposition der Eingabe zuweist.

In diesem Abschnitt werden wir unsere erste P-vollständige Sprache kennenlernen, das Circuit-Value-Problem (CVP). Das Problem CVP spielt als generisches Problem dieselbe Rolle wie das Erfüllbarkeitsproblem KNFSAT für die NP-Vollständigkeit, QBF für die PSPACE-Vollständigkeit oder D-REACHABILITY für die NL-Vollständigkeit.

Definition 6.11 *Wir nehmen an, daß der Fanin für alle zu betrachtenden Schaltkreise höchstens zwei ist.*

(a) Die Sprache des Circuit-Value Problem ist gegeben durch

$$\mathrm{CVP} = \{\langle S\rangle x \mid S \textit{ ist ein Schaltkreis mit Eingabe } x \textit{ und } S(x) = 1.\}$$

(b) Ein monotoner Schaltkreis besteht nur aus den Gatter $\wedge$ *und* $\vee$. *Die Sprache des monotonen Circuit-Value Problems ist gegeben durch*

$$\mathrm{M-CVP} = \{\langle S\rangle x \mid \textit{der monotone Schaltkreis } S \textit{ akzeptiert die Eingabe } x\ \}.$$

(c) Ein NOR-Schaltkreis besteht nur aus NOR-Gattern. Die Sprache des NOR-Circuit-Value Problems ist gegeben durch:

$$\mathrm{NOR-CVP} = \{\langle S\rangle x \mid \textit{der NOR-Schaltkreis } S \textit{ akzeptiert die Eingabe } x\ \}.$$

Lemma 6.12 *Die Sprachen* CVP, M − CVP *und* NOR − CVP *liegen in* P.

Beweis: Ein Schaltkreis der Größe s mit n Eingaben kann sequentiell zum Beispiel mit Hilfe der Tiefensuche in Zeit $O(s+n)$ ausgewertet werden. □

Das zentrale Resultat dieses Abschnitts ist die P-Vollständigkeit des Circuit-Value Problems.

Satz 6.13 CVP *ist* P*-vollständig.*

Beweis: Nach Lemma 6.12 ist CVP eine Sprache in P. Wir müssen noch zeigen, daß CVP P-hart ist. Sei L eine beliebige Sprache in P. Wir zeigen die Reduktion $L \leq_{\mathrm{LOG}}$ CVP.

Was ist über L bekannt? Es gibt eine deterministische Turingmaschine M, die L in höchstens $t(n)$ Schritten akzeptiert. Die Schrittzahl $t(n)$ ist durch ein Polynom $q(n)$ nach oben beschränkt. Um die geforderte Reduktion zu konstruieren, simulieren wir M für Eingaben der Länge n durch einen Schaltkreis S_n.

Wir simulieren M mit einem Schaltkreis S_n, dessen Grobstruktur einem zwei-dimensionalen Gitter entspricht. Die „i-te Zeile“ des Gitters gibt Bandinhalt, Kopfposition und Zustand von M zum Zeitpunkt i wieder. Die i-te Zeile ist aus identischen kleinen Schaltreisen $S_{i,j}$ aufgebaut, wobei $S_{i,j}$ die Zelle j zum Zeitpunkt i simuliert. Ein solcher kleiner Schaltkreis $S_{i,j}$ muß

- den von Schaltkreis $S_{i-1,j}$ berechneten Bandinhalt speichern können, falls der Kopf von M zum Zeitpunkt i die Zelle j nicht besucht und sonst
- den Bandinhalt verändern, abhängig vom gegenwärtigen Zustand und vom gegenwärtigen Bandinhalt. Weiterhin muss $S_{i,j}$ in diesem Fall den neuen Zustand und die Richtung des Kopfes festlegen.

Diese Aufgaben lassen sich durch einen Schaltkreis konstanter Größe bewerkstelligen, wenn wir die Ausgänge von $S_{i-1,j-1}$, $S_{i-1,j}$ und $S_{i-1,j+1}$ zu Eingängen von $S_{i,j}$ machen. Die Ausgänge jedes Schaltkreises müssen

- den Bandinhalt kodieren,
- angeben, ob die Zelle gerade besucht wurde und wenn ja, den neuen Zustand und die Kopfrichtung spezifizieren.

Beachte, dass sämtliche Schaltkreise $S_{i,j}$ „baugleich" gewählt werden können, mit Ausnahme der Schaltkreise $S_{0,j}$, die entweder zu setzen sind ($j \notin \{1,\ldots,n\}$) oder an die Eingabe anzuschliessen sind ($j \in \{1,\ldots,n\}$).

Wir müssen das Gitter noch „auswerten", d.h. wir müssen feststellen, ob der letzte Zustand akzeptierend ist. Dies gelingt, wenn wir einen binären Auswertungsbaum „auf" das Gitter setzen. Man beachte, daß der beschriebene Schaltkreis durch eine logarithmisch-platzbeschränkte Turingmaschine konstruierbar ist. □

Bemerkung 6.14 *Der Beweis von Satz 6.13 zeigt, dass wir eine polynomiell zeitbeschränkte Turingmaschine durch eine uniforme Schaltkreisfamilie polynomieller Größe simulieren können. Für den Nachweis von* P $\neq$ NP *genügt somit der Nachweis, dass irgendein* NP*-vollständiges Problem keine polynomiell großen Schaltkreise besitzt.*

Satz 6.15 M – CVP *und* NOR – CVP *sind* P*-vollständig.*

Beweis: Nach Lemma 6.12 sind M-CVP und NOR-CVP Sprachen aus P. Wir müssen noch zeigen, daß die Sprachen P-hart sind.

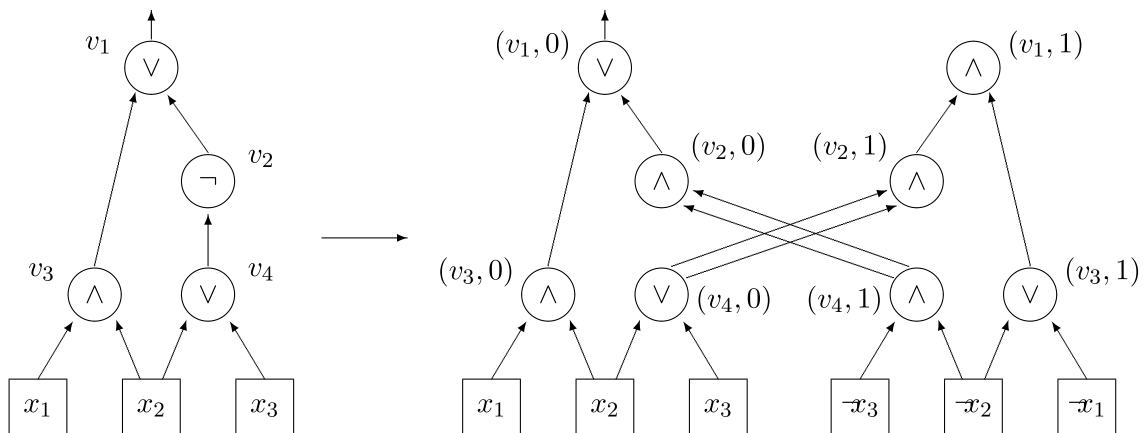


Abbildung 6.1: Beispiel zur Reduktion CVP $\leq_{\text{LOG}}$ M-CVP

Wir zeigen die Reduktion CVP $\leq_{\text{LOG}}$ M-CVP.

Sei (S, x) eine Eingabe für CVP. Wir „schieben“ sämtliche Negationsgatter hinab zu den Quellen von S. Da

$$\begin{aligned} \neg(u \wedge w) &\Leftrightarrow \neg u \vee \neg w \\ \neg(u \vee w) &\Leftrightarrow \neg u \wedge \neg w \\ \neg(\neg u) &\Leftrightarrow u \end{aligned}$$

bedeutet der Schiebeprozess das Vertauschen von $\wedge$- und $\vee$-Gattern sowie die Einführung neuer Quellen für die negierte Eingaben.

Wir ersetzen ein Gatter v durch zwei Gatter $(v, 0)$ und $(v, 1)$ mit den Ausgaben v beziehungsweise $\neg v$. Die obige Abbildung zeigt ein Beispiel. Formal sieht die Transformation wie folgt aus:

1. Für jedes Gatter v von S (mit Ausnahme der Senke von S) führen wir zwei Gatter $(v, 0)$ und $(v, 1)$ ein. Mit $(v, 0)$ simulieren wir das ursprüngliche Gatter, mit $(v, 1)$ simulieren wir das negierte Gatter. Die Senke s von S wird durch die Senke $(s, 0)$ des neuen Schaltkreises ersetzt. Wir nehmen an, daß die Senke ein $\wedge$- oder $\vee$-Gatter war.

2. Für jedes $\wedge$-, $\vee$- oder $\neg$-Gatter des alten Schaltkreises fügen wir im neuen Schaltkreis Verbindungen, wie in den beiden Abbildungen angedeutet, ein.

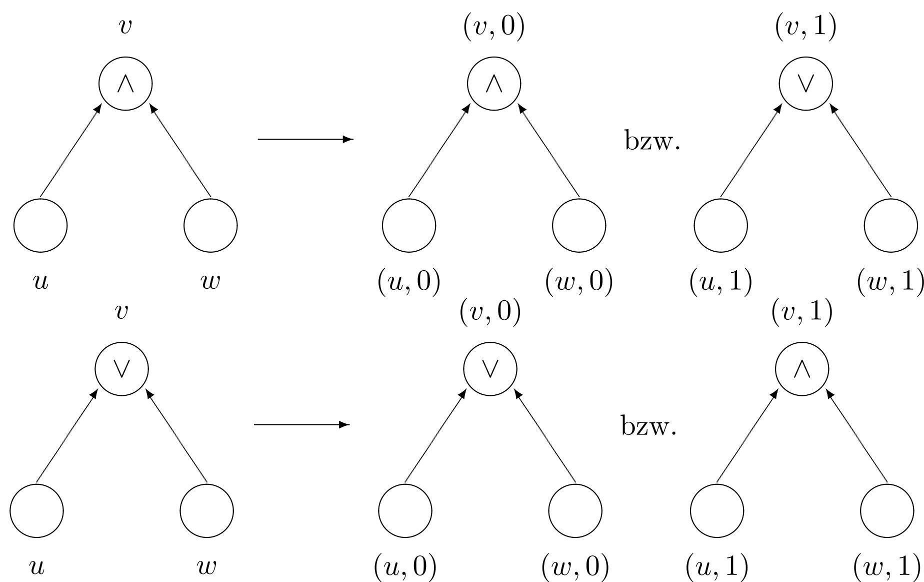


Abbildung 6.2: Ersetzen von $\wedge$- und $\vee$-Gattern.

3. Wir ersetzen eine Quelle v (mit Eingabe x_i) durch die beiden Quellen $(v, 0)$ (mit Eingabe x_i) und $(v, 1)$ (mit Eingabe $\neg x_i$).

Sei S^* der neue Schaltkreis (mit gleicher Senke wie S) und sei x^* die neue Eingabefolge von S^*. Offenbar gilt:

$$S \text{ akzeptiert } x \Leftrightarrow S^* \text{ akzeptiert } x^*.$$

Beachte, dass die Transformation

$$\langle S \rangle x \mapsto \langle S^* \rangle x^*$$

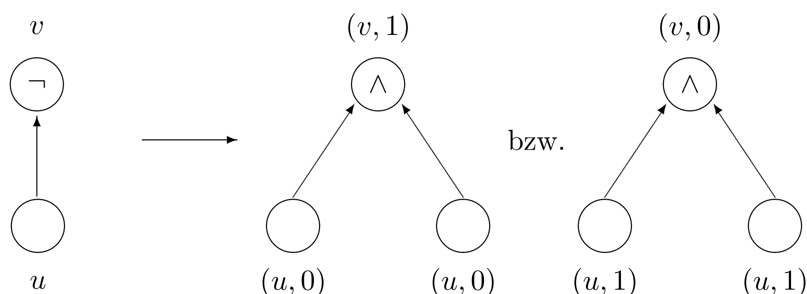


Abbildung 6.3: Ersetzen von ¬-Gattern.

durch eine Turingmaschine mit logarithmischer Platzbeschränkung berechnet werden kann.

Wir zeigen die Reduktion CVP $\leq_{\text{LOG}}$ NOR-CVP. Es ist $\text{nor}(u, v) = \neg(u \vee v)$. Jede der Operationen $\wedge$, $\vee$ und $\neg$ kann man mit NOR-Gattern darstellen, da $\text{nor}(u, u) = \neg u$, $\text{nor}(\neg u, \neg v) = u \wedge v$ und $\text{nor}(\text{nor}(u, v), \text{nor}(u, v)) = u \vee v$. Eine deterministische, logarithmisch-platzbeschränkte Turingmaschine kann somit zu jedem $\{\wedge, \vee, \neg\}$-Schaltkreis S einen äquivalenten NOR-Schaltkreis S^* konstruieren. Es gilt

$$S \text{ akzeptiert } x \Leftrightarrow S^* \text{ akzeptiert } x.$$

Beachte, dass die Reduktion

$$\langle S \rangle x \mapsto \langle S^* \rangle x$$

in logarithmischem Platz ausgeführt werden kann. □

6.2.2 Die Lineare Programmierung

Wir betrachten als nächstes das Problem der linearen Ungleichungen und der linearen Programmierung.

Definition 6.16 *Gegeben ist eine ganzzahlige Matrix $A \in F^{m \times n}$ sowie Vektoren $b \in \mathbb{Z}^m$ und $c \in \mathbb{Z}^n$.*

(a) Im Problem der linearen Ungleichungen ist zu entscheiden, ob es einen Vektor $x \in \mathbb{Q}^n$ mit $Ax \leq b$ gibt.

(b) Im Problem der linearen Programmierung ist zu entscheiden, ob es einen Vektor $x \in \mathbb{Q}^n$ mit $Ax \leq b$ und $c \cdot x \geq t$ gibt.

Das Problem der linearen Programmierung wird konventionell als Optimierungsproblem formuliert:

$$\text{Maximiere } c \cdot x, \text{ so daß } Ax \leq b \text{ und } x \geq 0 \text{ gilt.}$$

Wir hingegen haben dieses Optimierungsproblem als Entscheidungsproblem formuliert.

Satz 6.17

(a) Das Problem der linearen Ungleichungen ist `P`*-vollständig.*

(b) Das Problem der linearen Programmierung ist P*-vollständig.*

Beweis: Beide Sprachen liegen in P, denn es existieren Polynomialzeit-Algorithmen. Bekannte Beispiele für solche Verfahren sind Karmarkars Algorithmus und die Ellipsoid-Methode.

(a) Wir zeigen die Reduktion M-CVP$\leq_{\text{LOG}}$ Lineare Ungleichungen. Sei (S, x) eine Eingabe für M-CVP. Wir weisen jedem Gatter von S Ungleichungen zu.

- Beschreibe zuerst die Eingabe x.
  - Falls $x_i = 0$, verwende die Ungleichungen $x_i \leq 0$ und $-x_i \leq 0$.
  - Falls $x_i = 1$, verwende die Ungleichungen $x_i \leq 1$ und $-x_i \leq -1$.
- Für ein Gatter $v \equiv u \wedge w$ verwende die Ungleichungen: $v \leq u,\ v \leq w,\ u+w-1 \leq v,\ 0 \leq v$.
- Für ein Gatter $v \equiv u \vee w$ verwende die Ungleichungen: $u \leq v,\ w \leq v,\ v \leq u+w,\ v \leq 1$.

Durch Induktion über die topologische Nummer eines Gatters zeigt man, dass das lineare Ungleichungssystem genau eine Lösung hat und der gewünschten Lösung entspricht. Wir fügen noch die Ungleichung

$$-s \leq -1$$

für die eindeutig bestimmte Senke s des Schaltkreises hinzu. S akzeptiert genau dann die Eingabe x, wenn das konstruierte, lineare Ungleichungssystem lösbar ist.

(b) Die Behauptung folgt, da die Reduktion

$$\text{lineare Ungleichungen} \leq_{\text{LOG}} \text{lineare Programmierung}$$

trivial ist. □

Korollar 6.18 *Das Problem der linearen Ungleichungen und das Problem der linearen Programmierung bleiben sogar dann* P*-vollständig, wenn man die Koeffizienten der Matrix A und des Vektors b auf $\{-1, 0, 1\}$ beschränkt.*

6.2.3 Parallelisierung von Greedy-Algorithmen

Greedy-Methoden sind im allgemeinen leicht zu implementieren und führen zu effizienten, sequentiellen Algorithmen. Leider sind Greedy-Algorithmen häufig inhärent sequentiell. Wir werden am Beispiel einer Heuristik für das Independent-Set-Problem sehen, dass eine Parallelisierung nicht gelingen wird. (Gleichwohl gibt es andere Heuristiken, die lokale Minima in polylogarithmischer Zeit bestimmen.)

Sei $G = (V, E)$ ein ungerichteter Graph mit $V = \{1, \ldots, n\}$. Im Independent-Set Problem ist eine unabhängige Knotenmenge größter Kardinalität ist zu bestimmen. (Eine Knotenmenge ist unabhängig, wenn keine zwei Knoten der Menge durch eine Kante verbunden sind.)

Algorithmus 6.1 *Heuristik für das Independent-Set-Problem.*

Die Eingabe besteht aus einem ungerichteten Graphen $G = (V, E)$ mit $V = \{1, \ldots, n\}$.

(a) $I(G) := \emptyset$.

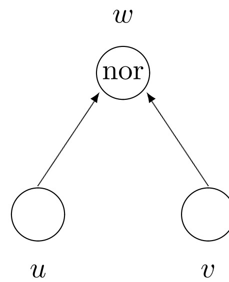


Abbildung 6.4: Nor-Gatter im Induktionsschritt zu Beweis 6.19

(b) FOR $v = 1$ TO n DO

IF (v ist nicht mit einem Knoten in $I(G)$ verbunden) THEN $I(G) = I(G) \cup \{v\}$.

(c) Die Menge $I(G)$ wird ausgegeben.

Die Heuristik des Greedy-Algorithmus' 6.1 findet in jedem Fall ein lokales Minimum (also eine unabhängige Menge, für die keine echte Obermenge unabhängig ist).

Im Lexicographically-First-Maximal-Independent-Set-Problem (LFMIS) ist ein ungerichteter Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$ und ein Knoten $v \in V$ gegeben. Es ist zu entscheiden, ob der Knoten v von Algorithmus 6.1 gewählt wird.

Satz 6.19 *Die Sprache LFMIS ist* `P`*-vollständig.*

Beweis: LFMIS liegt in `P`, denn die Laufzeit von Algorithmus 6.1 ist linear in n. Wir zeigen die Reduktion: NOR-CVP $\leq_{\text{LOG}}$LFMIS.

Sei (S, x) die Eingabe für NOR-CVP und sei $G = (V, E)$ die Graphstruktur des Schaltkreises S. Wir konstruieren einen ungerichteten Graphen $G^* = (V^*, E^*)$ aus dem gerichteten Graphen G: Füge einen neuen Knoten v_0 hinzu und setzen genau dann eine Kante $\{v_0, i\}$ zum Eingabeknoten i ein, wenn $x_i = 0$ ist. Es ist also

$$V^* = V \cup \{v_0\} \text{ und } E^* = \{\{v_0, i\} \mid x_i = 0\} \cup E.$$

Wir nummerieren die Knoten, so dass v_0 die Nummer Eins erhält und die übrigen Nummern der topologischen Nummerierung des Schaltkreises entsprechen. Durch Induktion über die Nummerierung zeigen wir

$$I(G^*) = \{v_0\} \cup \{v \in V \mid \text{Gatter } v \text{ hat den Wert } 1\}.$$

- Verankerung: Da der Algorithmus stets den ersten Knoten in $I(G^*)$ aufnimmt, ist $v_0 \in I(G)$. Wenn eine Quelle i den Wert Null hat, wird i wegen der Kante $\{v_0, i\}$ nicht in $I(G^*)$ aufgenommen. Wenn eine Quelle i den Wert Eins hat, wird i in $I(G^*)$ aufgenommen.

- Induktionsschritt: Betrachten wir den Knoten w, der im Schaltkreis einem Nor-Gatter $w =$nor(u, v) entspricht. Die Nummer der Knoten u und v ist kleiner als die Nummer von w und es ist $w = \neg(u \vee v) = 0 \leftrightarrow (u = 1) \;\vee\; (v = 1)$. Nach Induktionsannahme ist u (bzw. v) genau dann in $I(G^*)$, wenn $u = 1$ (bzw. $v = 1$) ist. Wegen der Kanten $\{u, w\}$ und $\{v, w\}$ nimmt der Algorithmus genau dann w in die Menge $I(G^*)$ auf, wenn das Gatter w den Wert Eins hat.

Die Menge $I(G^*)$ besteht also genau aus dem Knoten v_0 und allen Knoten mit Ausgabe Eins. Für die Senke s des NOR-Schaltkreises S ist

$$S(x) = 1 \Longleftrightarrow s \in I(G^*)$$

und dies war zu zeigen. □

6.3 Zusammenfassung

Wir haben Schaltkreise als paralleles Rechnermodell gewählt und die Klasse NC aller parallelisierbaren Sprachen eingeführt. Wir haben gesehen, dass ein enger Zusammenhang zwischen den Komplexitätsmaßen Tiefe und Speicherplatz besteht. Die *Parallel-Computation-Thesis* verallgemeinert diesen Zusammenhang auf alle möglichen Modelle paralleler Rechner und postuliert eine polynomielle Beziehung zwischen der Rechenzeit eines jeden vernünftigen parallelen Rechnermodells und der Speicherplatzkomplexität der berechneten Sprache.

Wir haben die Parallelisierbarkeit von Problemen untersucht und die im Hinblick auf eine Parallelisierung schwierigsten Sprachen, die P-vollständigen Sprachen mit Hilfe der LOGSPACE-Reduktion eingeführt. Wir haben das P-vollständige Circuit-Value Problem kennengelernt, das als generisches Problem dieselbe Rolle für die P-Vollständigkeit spielt wie das Erfüllbarkeitsproblem KNFSAT für die NP-Vollständigkeit, QBF für PSPACE oder wie D-REACHABILITY für die NL-Vollständigkeit. Die Lineare Programmierung wie auch die Bestimmung der lexikographisch ersten maximalen unabhängigen Menge sind weitere P-vollständige Probleme.