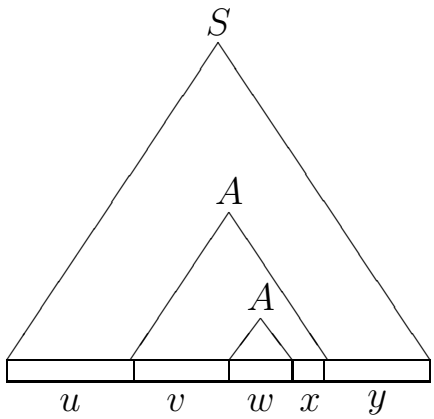
# Formale Grundlagen der Informatik

## - Vorlesungsskript -

Prof. Dr. Uwe Schöning




2012

# Inhaltsverzeichnis

# Teil 1

# Boolesche Funktionen, Schaltkreise und Codes

## 1.1 Boolesche Funktionen

Sei im Folgenden $\mathbb{B} = \{0, 1\}$. Eine *Boolesche Funktion* ist eine Funktion $f : \mathbb{B}^n \longrightarrow \mathbb{B}^m$. (Häufig ist $m = 1$, wir beschränken uns im Folgenden auch auf den Fall $m = 1$.) Wir beschreiben Boolesche Funktionen, indem wir sie aus einigen einfachen Grundfunktionen durch Verknüpfung zusammensetzen.

Die bekanntesten Grundfunktionen sind:

Die *Negationsfunktion*, bezeichnet mit $\neg$. Diese Funktion ist einstellig und durch folgende Tabelle definiert.

| $x$ | $\neg x$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

Anstelle des Zeichens $\neg$ überstreicht man auch häufig die zu negierende Funktion bzw. Formel.

Die *Oder-Funktion* (Disjunktion), symbolisch: $\vee$, ist zweistellig und durch folgende Verknüpfungstafel gegeben:

| $\vee$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

Die *Und-Funktion* (Konjunktion), symbolisch: $\wedge$, ist zweistellig und durch folgende Verknüpfungstafel gegeben:

| $\wedge$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Es gibt genau $2^{2^n}$ viele verschiedene $n$-stellige Boolesche Funktionen. (Begründung: der Definitionsbereich $\mathbb{B}^n$ hat $2^n$ viele Elemente; für jedes mögliche Argument aus $\mathbb{B}^n$ kann

eine Funktion den Wert 0 oder 1 annehmen; dies sind $2^{2^n}$ viele Möglichkeiten; also ist dies die Anzahl der verschiedenen $n$-stelligen Booleschen Funktionen). Bei $n > 5$ gibt es bereits mehr $n$-stellige Boolesche Funktionen als es Atome im Universum gibt. Zweistellige Boolesche Funktionen gibt es also 16 Stück. Wenn wir hierbei die "trivialen" Funktionen weglassen (solche die konstant auf 0 oder auf 1 abbilden und solche, die nur von einer Variablen abhängen), so bleiben noch 10 nicht-triviale übrig. Diese wiederum gliedern sich in zwei Gruppen mit je 5 Funktionen. Die eine Gruppe geht aus der anderen durch Negation hervor. Zwei der 5 Funktionen sind die Oder-Funktion und die Und-Funktion; die zugehörigen negierten Funktionen sind die Nor-Funktion (oder Peirce-Funktion) und die Nand-Funktion (oder Sheffer-Funktion) (Nor = not or, Nand = not and).

Die folgenden drei Funktionen kommen nun noch hinzu. Wir verwenden die Bezeichnungen $\rightarrow$ (Implikationsfunktion), $\leftarrow$ (die umgekehrte Implikation) und $\oplus$ (Xor-Funktion, exclusive or, ausschließendes Oder, Antivalenzfunktion, Negation der Äquivalenz). Diese sind durch folgende Verknüfungstafeln gegeben:

| $\rightarrow$ | 0 | 1 |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 1 |

| $\leftarrow$ | 0 | 1 |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 1 | 1 |

| $\oplus$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Wir geben die wichtigsten Gesetze im Umgang mit Booleschen Funktionen an:

**Berechnungen mit Konstanten**

$x \vee 0 = x, \quad x \vee 1 = 1, \quad x \wedge 0 = 0, \quad x \wedge 1 = x, \quad x \oplus 0 = x, \quad x \oplus 1 = \neg x$

**Assoziativität**

$x \vee (y \vee z) = (x \vee y) \vee z, \quad x \wedge (y \wedge z) = (x \wedge y) \wedge z, \quad x \oplus (y \oplus z) = (x \oplus y) \oplus z$

**Kommutativität**

$x \vee y = y \vee x, \quad x \wedge y = y \wedge x, \quad x \oplus y = y \oplus x$

**Distributivität**

$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z), \quad x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z), \quad x \wedge (y \oplus z) = (x \wedge y) \oplus (x \wedge z)$

**Vereinfachungsgesetze**

$x \vee x = x, \quad x \vee \neg x = 1, \quad x \wedge x = x, \quad x \wedge \neg x = 0, \quad x \oplus x = 0, \quad x \oplus \neg x = 1$

**Absorptionsgesetze**

$x \vee (x \wedge y) = x, \quad x \wedge (x \vee y) = x$

**deMorgansche Gesetze, Negationsgesetze**

$\neg x \vee \neg y = \neg(x \wedge y), \quad \neg x \wedge \neg y = \neg(x \vee y), \quad \neg x \oplus \neg y = x \oplus y, \quad \neg\neg x = x$

**Ersetzen der Implikation, Antivalenz und der Äquivalenz**

$x \to y = \neg x \vee y, \quad x \leftarrow y = x \vee \neg y,$
$x \oplus y = (\neg x \wedge y) \vee (x \wedge \neg y) = (\neg x \vee \neg y) \wedge (x \vee y)$
$x \leftrightarrow y = (x \wedge y) \vee (\neg x \wedge \neg y) = (\neg x \vee y) \wedge (x \vee \neg y)$

Die obigen Gesetze bleiben auch korrekt, wenn wir die Booleschen Variablen $x, y, z$ durch Boolesche Funktionen ersetzen.

Wir bemerken, dass die Struktur $(\mathbb{B}, \wedge, \vee)$ eine Boolesche Algebra (also ein distributiver, komplementärer Verband) ist, und dass $(\mathbb{B}, \oplus, \wedge)$ ein endlicher Körper, das Galois-Feld $GF(2)$, ist.

Anstelle von $x \wedge y$ werden wir in Zukunft oft $xy$ schreiben.

## 1.2 Normalformen

Wie kann man Boolesche Funktionen beschreiben? Eine Möglichkeit ist durch vollständiges Auflisten der Funktion in Form einer *Wahrheitstafel:*

*Beispiel:*

| $x$ | $y$ | $z$ | $f(x,y,z)$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Bei diesem Beispiel gibt es 3 Zeilen, die den Funktionswert 1 liefern. Offensichtlich ist der Wert der Funktion genau dann 1, wenn dies aufgrund einer dieser 3 Zeilen der Fall ist, d.h. wenn entweder

$$\begin{aligned} x = 0, \quad y = 0, \quad z = 0 \quad &\text{oder} \\ x = 0, \quad y = 1, \quad z = 1 \quad &\text{oder} \\ x = 1, \quad y = 1, \quad z = 1 \end{aligned}$$

Dies wiederum ist genau dann der Fall, wenn $\overline{x} \wedge \overline{y} \wedge \overline{z} = 1$ oder $\overline{x} \wedge y \wedge z = 1$ oder $x \wedge y \wedge z = 1$. Das bedeutet, wir können die Funktion schreiben als

$$f(x,y,z) = \overline{x}\,\overline{y}\,\overline{z} \ \vee \ \overline{x}\,y\,z \ \vee \ x\,y\,z$$

Offensichtlich kann mit dieser Methode jede Boolesche Funktion in eine solche Formel überführt werden. Im allgemeinen ist dies eine Formel der Art

$$\bigvee_{(a_1,\ldots,a_n)\in f^{-1}(1)} x_1^{a_1}\, x_2^{a_2}\, \ldots\, x_n^{a_n}$$

wobei $x^1 = x$ und $x^0 = \overline{x}$ bedeuten soll. Ein Ausdruck der Gestalt $x_1^{a_1}\, x_2^{a_2}\, \ldots\, x_n^{a_n}$ nennt man *Minterm*. Der Ausdruck Minterm (oder ausführlicher: Minimalterm) kommt daher, dass nur eine einzige Belegung der Variablen, nämlich $(a_1, \ldots, a_n)$, den Wert 1 liefert.

Wenn wir bei der Erstellung dieser Formel systematisch die Wahrheitstafel von oben nach unten und innerhalb einer Zeile mit Funktionswert 1 von links nach rechts vorgehen, so ist diese Formel sogar eindeutig bestimmt. Wir nennen dies die (vollständige) *disjunktive Normalform* (kurz: DNF) von $f$.

Ist $f$ durch eine Formel gegeben, so kann man auch – ohne Aufstellen einer Wahrheitstafel – zur DNF von $f$ gelangen, indem man die obigen Regeln in systematischer Form anwendet:

1. Zunächst in eine Formel, in der nur die Booleschen Operationen $\vee, \wedge, \neg$ vorkommen, umformen.
2. Mittels der deMorganschen Regeln können alle Negationszeichen bis direkt vor die Variablen gebracht werden.
3. Durch (evtl. mehrfaches) Anwenden des Distributivgesetzes können Und-Verknüpfungen, die auf Oder-Verknüpfungen angewandt werden, in Oder-Verknüpfungen, die auf Und-Verknüpfungen angewandt werden, umgeformt werden.

Jetzt erhalten wir eine (mehrfache) Disjunktion von Konjunktionen; im Innern der Konjunktionen befinden sich nur Variablen und negierte Variablen. Allerdings kommen in den Konjunktionen nicht unbedingt alle $n$ Variablen vor (verkürzte DNF). Diese kann durch systematisches Einführen aller fehlenden Variablen dann in DNF gebracht werden. (Beispiel: aus $y\, z$ wird $x\, y\, z \;\vee\; \overline{x}\, y\, z$).

Alle Gesetze für Boolesche Funktionen gelten auch in der dualen Form, das heißt, man vertausche bei einer Regel 0 mit 1, $x$ mit $\overline{x}$, $\vee$ mit $\wedge$, und umgekehrt, und man erhält wieder eine Regel für Boolesche Funktionen. Indem man in dieser Weise alle Anweisungsschritte zur Erzeugung der DNF "verdreht", erhält man eine Anleitung zur Herstellung der *konjunktiven Normalform*, kurz: KNF. Die allgemeine Form für die KNF ist

$$\bigwedge_{(a_1,\ldots,a_n)\in f^{-1}(0)} (x_1^{\overline{a_1}} \vee x_2^{\overline{a_2}} \vee \ldots \vee x_n^{\overline{a_n}})$$

Einen Ausdruck der Form $(x_1^{\overline{a_1}} \vee x_2^{\overline{a_2}} \vee \ldots \vee x_n^{\overline{a_n}})$ nennt man *Maxterm*.

Bei der obigen Wahrheitstafelmethode orientiere man sich also an den Zeilen, die den Funktionswert 0 ergeben, und erhält so:

$$f(x,y,z) = (x \vee y \vee \overline{z})\,(x \vee \overline{y} \vee z)\,(\overline{x} \vee y \vee z)\,(\overline{x} \vee y \vee \overline{z})\,(\overline{x} \vee \overline{y} \vee z)$$

Beachte: Eine Funktion, deren DNF aus $k$ Mintermen besteht, besitzt in in der KNF-Darstellung $2^n - k$ Maxterme (und umgekehrt).

Was wir in diesem Abschnitt bewiesen haben ist der folgende Satz.

**Satz.**
JEDE BOOLESCHEN FUNKTION LÄSST SICH SOWOHL IN DISJUNKTIVER NORMALFORM (ALS DISJUNKTION VON MINTERMEN) AUCH IN KONJUNKTIVER NORMALFORM (ALS KONJUNKTION VON MAXTERMEN) DARSTELLEN.

## 1.3 Minimierung von Normalformen

Die DNF und KNF einer Formel sind im Allgemeinen recht große Formeln; und zwar in gewisser Weise größer als sie sein müssten. Betrachten wir zum Beispiel die folgende Formel in DNF

$$f(x,y,z) = x\,y\,z \vee \overline{x}\,y\,z \vee \overline{x}\,\overline{y}\,\overline{z}$$

Hier können die ersten 2 Terme vereinfacht werden, indem $x$ eliminiert wird:

$$f(x,y,z) = y\,z \vee \overline{x}\,\overline{y}\,\overline{z}$$

Dies ist wieder eine Formel in DNF (nämlich eine Disjunktion von Konjunktionen), aber nicht mehr die vollständig ausgeschriebene, sondern eine vereinfachte. Diese Vereinfachungen können sehr weitgehend sein. Hier ist ein Beispiel:

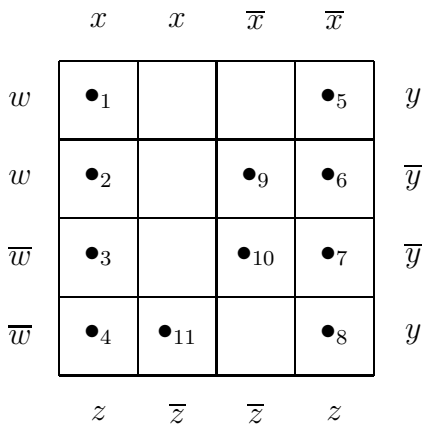
$$f(w,x,y,z) = w\,x\,y\,z \vee w\,x\,\overline{y}\,z \vee \overline{w}\,x\,\overline{y}\,z \vee \overline{w}\,x\,y\,z \vee w\,\overline{x}\,y\,z \vee w\,\overline{x}\,\overline{y}\,z$$

$$\vee\ \overline{w}\,\overline{x}\,\overline{y}\,z \vee \overline{w}\,\overline{x}\,y\,z \vee w\,\overline{x}\,\overline{y}\,\overline{z} \vee \overline{w}\,\overline{x}\,\overline{y}\,\overline{z} \vee \overline{w}\,x\,y\,\overline{z}$$

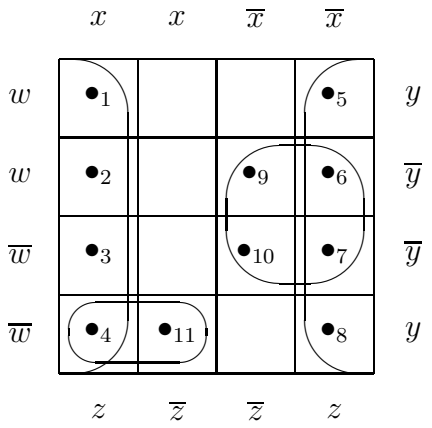
Diese DNF lässt sich vereinfachen bis zu

$$f(w,x,y,z) = z \vee \overline{x}\,\overline{y} \vee \overline{w}\,x\,y$$

Wie macht man das? Bei wenigen (das heißt: bis zu 4 oder 5) Variablen kann man dies graphisch durch sog. *Karnaugh-Veitch-Diagramme* (kurz: KV-Diagramme) machen: Wir tragen alle möglichen Minterme der DNF (bzw. Wertzuweisungen an die Variablen, die den Wert 1 ergeben) in ein zweidimensionales Diagramm ein:



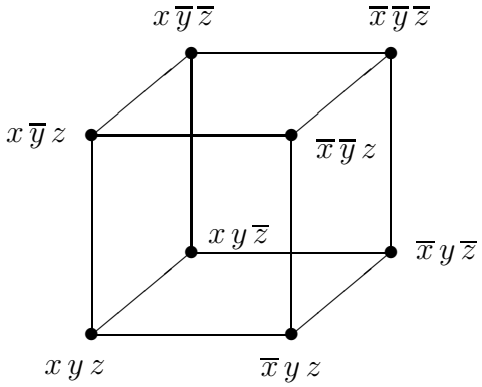
Wir haben hier die Minterme, die in obiger Beispielfunktion $f$ vorkommen, mit einem Punkt markiert (und zusätzlich noch durchnummeriert). Indem man nun die Bereiche mit Punkten (die Minterme) durch zusammenhängende 8er-, 4er-, 2er oder 1er-Blöcke zusammenfasst, erhält man eine vereinfachte DNF. Hierbei dürfen sich die Blöcke auch überlappen. Wichtig ist, dass man möglichst große und möglichst wenige Blöcke auswählt. Blöcke dürfen auch über die Kanten hinweg ge“wrapped” werden. Ein 4er Block muss entweder ein Quadrat oder eine waagerechte oder senkrechte Reihe aus 4 Elementen bilden. Durch das “wrapping” bilden z.B. auch die 4 Elemente, die an den 4 Ecken des KV-Diagramms sitzen, einen Block. Ein 8er-Block entspricht dann einem Ausdruck mit einer Variablen, ein 4er-Block entspricht einem Ausdruck mit zwei Variablen, ein 2er-Block entspricht einem Ausdruck mit drei Variablen und ein 1er-Block entspricht einem Ausdruck mit vier Variablen (also einem Minterm).




Diese Überdeckung ergibt die oben angegebene vereinfachte Formel: Die Minterme $\{1, 2, 3, 4, 5, 6, 7, 8\}$ ergeben $z$; die Minterme $\{6, 7, 9, 10\}$ ergeben $\overline{x}\,\overline{y}$; und $\{4, 11\}$ ergibt $\overline{w}\,x\,y$.

Diese zweidimensionale Darstellung in Form einer KV-Tafel ist nur eine Hilfsvorstellung; tatsächlich müsste man bei $n$ Variablen einen $n$-dimensionalen “Würfel” verwenden. Mit einer Kante verbunden sind immer solche Terme, die sich nur in einer Variablen unterscheiden (diese kommt im einen Term positiv, im anderen negiert vor).

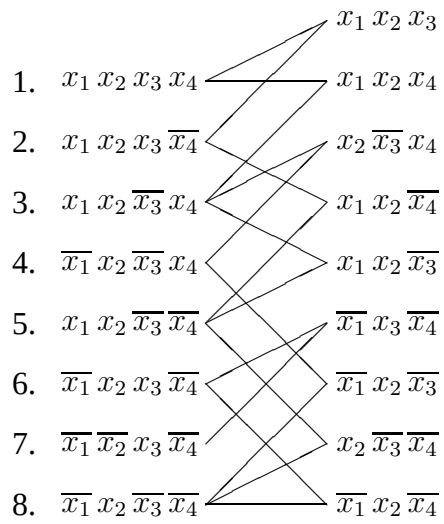
*Beispiel,* $n = 3$*:*



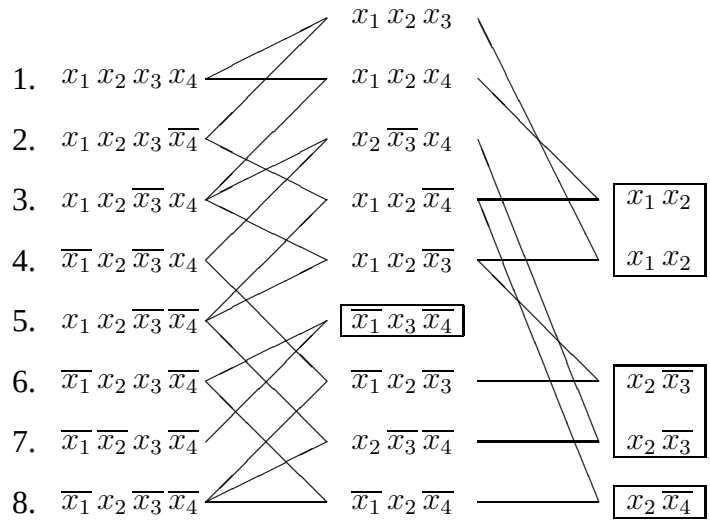

In der Literatur wird ein auf den Computer zugeschnittenes Verfahren zur Minimierung der DNF oft angegeben, das man auch noch anwenden kann bei einer größeren Anzahl von Variablen. (Allerdings auch nur bis zu einem gewissen Grad; das Verfahren wird auch ziemlich bald an der exponentiellen Rechenzeit scheitern). Dies ist das *Quine-McCluskey Verfahren*. Gegeben sei die zu minimierende vollständige DNF (oder die Wahrheitstafel der Funktion, anhand der man alle Minterme der vollständigen DNF ablesen kann). Wir schreiben diese Minterme (bei unserem Beispiel mit 4 Variablen) untereinander und nummerieren sie durch:

1. $x_1\,x_2\,x_3\,x_4$
2. $x_1\,x_2\,x_3\,\overline{x_4}$
3. $x_1\,x_2\,\overline{x_3}\,x_4$
4. $\overline{x_1}\,x_2\,\overline{x_3}\,x_4$
5. $x_1\,x_2\,\overline{x_3}\,\overline{x_4}$
6. $\overline{x_1}\,x_2\,x_3\,\overline{x_4}$
7. $\overline{x_1}\,\overline{x_2}\,x_3\,\overline{x_4}$
8. $\overline{x_1}\,x_2\,\overline{x_3}\,\overline{x_4}$

Dann untersuchen wir systematisch alle Paare von Mintermen, und finden die heraus, die sich nur im “Vorzeichen” einer einzigen Variablen unterscheiden. Wir erzeugen bei Vorliegen dieser Situation einen sog. *Resolventen*, der durch Streichen dieser Variablen entsteht. Auf diese Weise entsteht eine Folge von Resolventen der gegebenen Minterme, die wir rechts daneben notieren. Ferner vermerken wir durch Kanten, durch welche Minterme die Resolventen erzeugt wurden.




Wir fahren nun genauso bei den Resolventen fort und erzeugen die nächste Spalte, also die Resolventen der Resolventen; und solange dies geht, immer so weiter.




Das Verfahren bricht natürlich irgendwann ab (bei unserem Beispiel: jetzt). Nun notieren wir alle Minterme und Resolventen, die keine Nachfolger haben (von denen keine Kante nach rechts ausgeht). Diese sind oben durch Rechtecke gekennzeichnet. Es kann ferner wie man sieht vorkommen, dass manche der Resolventen, die über verschiedene Kantenwege zu Stande kommen, identisch sind. Diese übrigbleibenden Ausdrücke in den Rechtecken heißen *Primimplikanten*. Die Oder-Verknüpfung aller Primimplikanten wäre bereits ein Ergebnis, das äquivalent zur ursprünglichen Formel ist; aber dieses ist noch nicht unbedingt minimal, denn einige Primimplikanten können evtl. weggelassen werden. Hierzu ordnen wir die Primimplikanten in eine Zuordnungstabelle ein, die uns Auskunft darüber gibt, welche der ursprünglichen Minterme durch welche Primimplikanten “abge-

deckt" werden. (Dies ist dann der Fall, wenn von dem betreffenden Minterm eine Kante zu dem Primimplikanten führt).

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $\overline{x_1}\, x_3\, \overline{x_4}$ | | | | | | + | + | |
| $x_1\, x_2$ | + | + | + | | + | | | |
| $x_2\, \overline{x_3}$ | | | + | + | + | | | + |
| $x_2\, \overline{x_4}$ | | + | | | + | + | | + |

Nun versuchen wir, eine minimale Auswahl aus den Primimplikaten zu treffen, so dass immer noch alle Minterme erfasst werden. Der Primimplikant $\overline{x_1}\, x_3\, \overline{x_4}$ muss auf jeden Fall bei unserer Minimalauswahl dabei sein, da nur er den Minterm 7 erfassen kann. Mit dem analogen Argument müssen die Primimplikanten $x_1\, x_2$ und $x_2\, \overline{x_3}$ ausgewählt werden, da sie als Einzige die Minterme 1 bzw. 4 erfassen können. Nach Auswahl dieser 3 Primimplikaten ergibt sich die Situation, dass bereits alle Minterme erfasst sind, es also nicht notwendig ist, den Primimplikant $x_2\, \overline{x_4}$ noch hinzuzunehmen.

Als Ergebnis erhalten wir die folgende vereinfachte DNF:

$$\overline{x_1}\, x_3\, \overline{x_4} \;\vee\; x_1\, x_2 \;\vee\; x_2\, \overline{x_3}$$

Die Aufgabe, aus der Primimplikantentabelle eine minimale Überdeckung aller Minterme zu bestimmen, ist keineswegs trivial: sie ist "NP-vollständig" (vgl. Komplexitätstheorie). Das bedeutet nach heutigem Stand des Wissens, dass kein wesentlich besserer als der "probier-alle-Möglichkeiten" Algorithmus zur Verfügung steht, und dies kostet exponentiell viel Rechenzeit.

Eine weitere algorithmische Verbesserung lässt sich erzielen, indem man die Minterme (und im weiteren Verlauf des Verfahrens, die Resolventen) entsprechend der Anzahl der vorkommenden negierten Variablen in Gruppen anordnet: Gruppe $i$ soll also genau diejenigen Minterme enthalten, in denen $i$ negative Variablen vorkommen. Dann braucht man bei der Konstruktion der Resolventen nur Minterme in Betracht ziehen, so dass der eine aus Gruppe $i$ stammt und der andere aus Gruppe $i+1$ (denn nur die können sich potenziellerweise im Vorzeichen von genau einer Variablen unterscheiden).

## 1.4 Vollständige Basen

Da es für jede Boolesche Funktion $f$ eine DNF (bzw. KNF) Formel gibt, und diese nur mittels $\vee, \wedge, \neg$ aufgebaut ist, heißt das, dass mittels dieses Operatorensatzes *alle* Booleschen Funktionen ausgedrückt werden können. Wir sagen, dass $\{\vee, \wedge, \neg\}$ eine *vollständige Basis* darstellt. Weitere vollständige Basen sind:

$\{\wedge, \neg\}$ : Die fehlende Operation $\vee$ kann ausgedrückt werden mittels: $x \vee y = \neg(\neg x \wedge \neg y)$.

$\{\vee, \neg\}$ : Die fehlende Operation $\wedge$ kann ausgedrückt werden mittels: $x \wedge y = \neg(\neg x \vee \neg y)$.

$\{\mathit{nand}\}$ : Es gilt $\neg x = \mathit{nand}\,(x, x)$ und $x \vee y = \mathit{nand}\,(\mathit{nand}\,(x, x), \mathit{nand}\,(y, y))$.

$\{\mathit{nor}\}$ : Es gilt $\neg x = \mathit{nor}\,(x, x)$ und $x \wedge y = \mathit{nor}\,(\mathit{nor}\,(x, x), \mathit{nor}\,(y, y))$.

$\{\oplus, \wedge\}$ : Es gilt $\neg x = 1 \oplus x$. (Die Konstanten 0 und 1 werden "kostenlos" zur Verfügung gestellt und werden nicht als Bestandteil der Basis angesehen).

$\mathbb{B}_2$ : Dies ist die Menge aller 2-stelligen Booleschen Funktionen (davon gibt es 16 Stück). $\mathbb{B}_2$ ist trivialerweise eine vollständige Basis und enthält alle oben angegebenen als Teilmenge.

Eine weitere interessante vollständige Basis stellt $\{\mathit{sel}\}$ dar. Dies ist eine 3-stellige Funktion, die wie folgt definiert werden kann:

$$\begin{aligned} \mathit{sel}\,(0, y, z) &= y \\ \mathit{sel}\,(1, y, z) &= z \end{aligned}$$

Je nach Wert des ersten Argument wird also das zweite oder das dritte Argument selektiert und zum Ausgang "durchgeschaltet". (In der technischen Informatik würde man sagen: ein *Multiplexer*). Die Vollständigkeit ergibt sich wie folgt:

$$x \vee y = \mathit{sel}\,(x, y, 1), \quad x \wedge y = \mathit{sel}\,(x, 0, y), \quad \neg x = \mathit{sel}\,(x, 1, 0)$$

Wir beobachten für spätere Verwendung, dass für jede Funktion $f$ gilt:

$$f(x_1, x_2, \ldots, x_n) = \mathit{sel}\,(x_1, f(0, x_2, \ldots, x_n), f(1, x_2, \ldots, x_n))$$

Dies nennt man auch „Shannon-Zerlegung".

## 1.5 Ringsummennormalform

Unter Zuhilfenahme der Basis $\{\oplus, \wedge\}$ ergibt sich eine weitere Normalform, die *Ringsummennormalform* (oder Reed-Muller-Entwicklung oder Ringsummenexpansion). Wir verwenden die Abkürzung RSNF). Diese wird im folgenden Satz eingeführt.

**Satz.**
FÜR JEDE BOOLESCHE FUNKTION $f : \mathbb{B}^n \longrightarrow \mathbb{B}$ GIBT ES GENAU EINEN 0-1-VEKTOR $a = (a_T)_{T \subseteq \{1,\ldots,n\}}$ SO DASS

$$f(x_1, \ldots, x_n) = \bigoplus_{T \subseteq \{1,\ldots,n\}} \left( a_T \wedge \bigwedge_{i \in T} x_i \right)$$

*Beweis:* Wir zeigen zunächst die Existenz eines solchen Vektors $a$. Weiter oben wurde argumentiert, dass die Basis $\{\oplus, \wedge\}$ vollständig ist. Es gibt also eine Formeldarstellung für $f$, in der nur $\oplus$ und $\wedge$ als Operatoren, die Variablen $x_1, \ldots, x_n$, sowie ggfs. die Konstanten 0 und 1 vorkommen. Nun wenden wir solange das Distributivgesetz für $\{\oplus, \wedge\}$ an (d.h. "Ausmultiplizieren"), wie dies möglich ist. Wir müssen evtl. auch Simplifikationen wie $x \oplus x = 0$, $0 \oplus x = x$, $0 \wedge x = 0$ und $1 \wedge x = x$ verwenden. Am Ende erhalten wir ein multilineares Polynom (über dem Körper GF(2)) in den Variablen $x_1, \ldots, x_n$. Der gesuchte Vektor $a$ ergibt sich daraus, welche der potenziellen $2^n$ vielen Ausdrücke der Form $\bigwedge_{i \in T} x_i$ in diesem Polynom vorkommen.

Als zweites zeigen wir die Eindeutigkeit der RSNF. Da die Anzahl der verschiedenen $n$-stelligen Booleschen Funktionen $f$ (nämlich $2^{2^n}$) genau der Anzahl der verschiedenen Vektoren $a = (a_X)_{X \subseteq \{1,\ldots,n\}}$ entspricht (denn diese 0-1-Vektoren haben die Länge $2^n$), und da *jeder* der $2^{2^n}$ vielen Booleschen Funktionen aufgrund des ersten Teils des Beweises *mindestens* eine RSNF zugeordnet werden kann, kann es umgekehrt aber keine zwei verschiedene RSNFen geben, die ein und der selben Booleschen Funktion zugeordnet sind. Daher ist die Zuordnung von $n$-stelligen Booleschen Funktionen zu Formeln in RSNF eine bijektive Abbildung. Die RSNF ist daher eindeutig. □

*Beispiel:*

$$\begin{aligned} f &= (\neg x \vee y) \wedge \neg z \\ &= \neg(x \wedge \neg y) \wedge \neg z \\ &= (1 \oplus (x\,(1 \oplus y)))\,(1 \oplus z) \\ &= (1 \oplus (x \oplus x\,y))\,(1 \oplus z) \\ &= 1 \oplus x \oplus z \oplus x\,y \oplus x\,z \oplus x\,y\,z \end{aligned}$$

Als Vektor $a = (a_X)_{X \subseteq \{1,\ldots,n\}}$ ergibt sich:

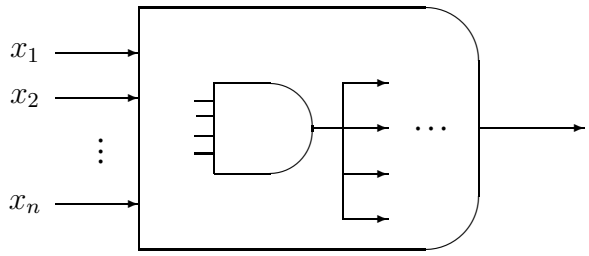
$$\begin{aligned} a_\emptyset &= 1 \\ a_{\{x\}} &= 1 \\ a_{\{y\}} &= 0 \\ a_{\{z\}} &= 1 \\ a_{\{x,y\}} &= 1 \\ a_{\{x,z\}} &= 1 \\ a_{\{y,z\}} &= 0 \\ a_{\{x,y,z\}} &= 1 \end{aligned}$$

## 1.6 Schaltkreise

Im allgemeinen erfordert es exponential in $n$ viele Terme, um eine Formel in KNF, DNF oder RSNF aufzuschreiben. Wir wollen uns als nächstes daher nach einer Darstellungs- und auch Berechnungsmöglichkeit für Boolesche Funktionen umsehen, die evtl. weniger als exponentialen Aufwand erfordert. In einer Formel könnten unter Umständen viele gleiche Teilformel vorkommen; diese müssten alle explizit hingeschrieben (und separat

evaluiert) werden, und dies ist ein größerer Aufwand. In einem *Schaltkreis* (auch Schaltnetz oder Boolesches Netzwerk genannt) dagegen könnte ein Teilschaltkreis vorkommen, der eine Funktion berechnet, die an verschiedenen Stellen wieder verwendet wird – und zwar dadurch, dass der Ausgang dieses Teilschaltkreis an meherere Stellen „weiterverdrahtet“wird. Dies erspart sowohl Schreibarbeit, als auch Rechenzeit (wenn der Schaltkreis tatsächlich eine Rechnung durchführen soll).

*Skizze:*




*Definition:* Ein Schaltkreis über einer Basis $\Omega$ von Grundfunktionen (zum Beispiel $\Omega = \{\vee, \wedge, \neg\}$) ist ein gerichteter, azyklischer Graph. Die Eingangsknoten sind mit Variablennamen oder den Konstanten 0,1 beschriftet, während die inneren Knoten mit Elementen aus $\Omega$ beschriftet sind. Die Anzahl der Vorgänger eines inneren Knotens muss mit der Stelligkeit der Grundfunktion übereinstimmen, mit der er beschriftet ist. Die inneren Knoten eines Schaltkreises nennen wir auch *Gatter*.

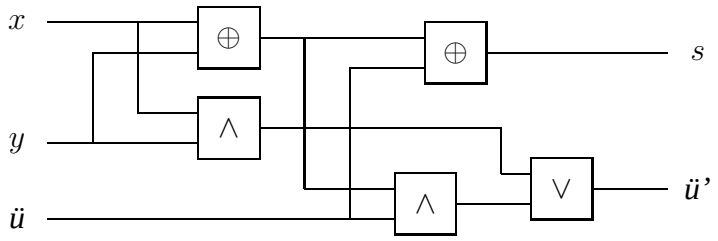
Wir definieren induktiv über die Tiefe des Schaltkreises die an einem Knoten $g$ berechnete Funktion $res_g(x_1, \ldots, x_n)$:

- Falls $g$ ein Eingangsknoten ist, der mit der Variablen 0 (bzw. 1) beschriftet ist, so ist $res_g(x_1, \ldots, x_n) = 0$ (bzw. $= 1$).
- Falls $g$ ein Eingangsknoten ist, der mit der Variablen $x_i$ beschriftet ist, so ist $res_g(x_1, \ldots, x_n) = x_i$.
- Falls $g$ ein innerer Knoten ist, der mit $f \in \Omega$ beschriftet ist, und falls $k$ die Stelligkeit von $f$ ist, so seien $f_1, \ldots, f_k$ die an den Vorgängerknoten von $g$ berechneten Funktionen. Dann ist

$$res_g(x_1, \ldots, x_n) = f(f_1(x_1, \ldots, x_n), \ldots, f_k(x_1, \ldots, x_n))$$

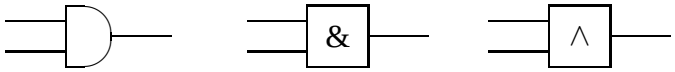
Wir sagen, dass ein Schaltkreis $S$ eine Boolesche Funktion $f$ *berechnet*, falls es einen Knoten $g$ in $S$ gibt mit $f = res_g$.

*Beispiel:* Der folgende Schaltkreis (basierend auf Gattern für die $\oplus$, $\wedge$, $\vee$ Funktion) ist eine Realisierung eines *Volladdierers*, das heißt, am Ausgang $s$ wird die Funktion $x \oplus y \oplus$ *ü* berechnet, während am Ausgang *ü'* die Funktion $(x \wedge y) \vee (ü \wedge (x \oplus y))$ berechnet wird.

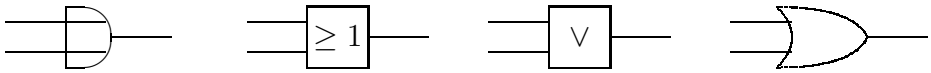



Für das Zeichnen von Schaltkreisen gibt es verschiedene Konventionen, bis hin zu DIN-Normen. Hier sind einige gebräuchliche Notationen:

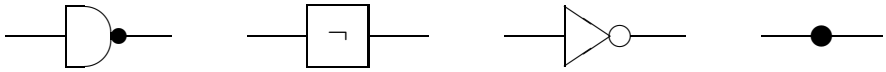
für das Und-Gatter:




für das Oder-Gatter:




für das Nicht-Gatter:




Die *Größe* eines Schaltkreises ist die Anzahl seiner Gatter. Die *Tiefe* eines Schaltkreises ist die Länge eines längsten Pfades im Schaltkreis. Die Größe ist ein Maß für den Verdrahtungs- und Materialaufwand. Die Tiefe ist ein Maß für die Signallaufzeit und damit die Schnelligkeit eines Schaltkreises. Ideal wären Schaltkreise mit polynomialer (womöglich sogar linearer) Größe und logarithmischer Tiefe. Tatsächlich hat die Klasse der Booleschen Funktionen, die durch Schaltkreise der Größe $n^k$ ($k$ konstant) und Tiefe $(\log n)^i$ berechenbar sind, in der Literatur den Namen $NC^i$ erhalten ($NC$ = "Nick's Class", nach dem Wissenschaftler Nicholas Pippenger). Von besonderem Interesse ist die Klasse $NC^1$. Für verschiedene, in der Praxis wichtige, Boolesche Funktionen (zum Beispiel für die Addition von 2 Binärzahlen, die Multiplikation, u.a.) gibt es derartige "effiziente" $NC^1$ Schaltkreise. (Vgl. I. Wegener: Effiziente Algorithmen für grundlegende Funktionen, Teubner, 1989).

*Bemerkung:* Wenn wir hier über "Boolesche Funktionen" reden und das asymptotische Verhalten von Schaltkreisen für diese Funktionen ansprechen (zum Beispiel polynomial oder logarithmisch), so ist eigentlich nicht eine einzelne Boolesche Funktion mit einer bestimmten Stelligkeit $n$ gemeint, sondern eine *Familie* von Funktionen "derselben Art": $\{f_n\}_{n\in\mathbb{N}}$, wobei $f_n : \mathbb{B}^n \longrightarrow \mathbb{B}$. Beispiel: "die" Paritätsfunktion ist die Familie aller Funktionen der Art $f_n(x_1, \ldots, x_n) = x_1 \oplus \ldots \oplus x_n$.

Es bezeichne $C(S)$ die Größe des Schaltkreises $S$ und $D(S)$ seine Tiefe. Für eine Boolesche Funktion $f$ ist $C_\Omega(f)$ die kleinstmögliche Größe eines Schaltkreises über der Basis $\Omega$, der $f$ berechnet. In ähnlicher Weise ist $D_\Omega(f)$ die kleinstmögliche Tiefe eines Schaltkreises, der $f$ berechnet:

$$\begin{aligned} C_\Omega(f) &= \min\{C(S) \mid S \text{ ist ein } \Omega\text{-Schaltkreis, der } f \text{ berechnet}\} \\ D_\Omega(f) &= \min\{D(S) \mid S \text{ ist ein } \Omega\text{-Schaltkreis, der } f \text{ berechnet}\} \end{aligned}$$

Wir werden zeigen, dass diese Komplexitätsmaße nicht wesentlich von der verwendeten Basis $\Omega$ abhängen, solange diese vollständig ist (d.h. wenn alle Booleschen Funktionen mit Grundfunktionen aus $\Omega$ berechnet werden können).

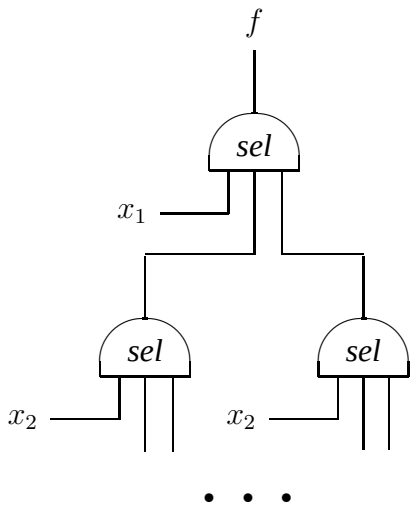
**Satz.**
SEIEN $\Omega$ UND $\Omega'$ ZWEI ENDLICHE, VOLLSTÄNDIGE BASEN. SEI $c = \max\{C_{\Omega'}(g) \mid g \in \Omega\}$ UND SEI $d = \max\{C_\Omega(g) \mid g \in \Omega'\}$. DANN IST FÜR ALLE BOOLESCHEN FUNKTIONEN $f$, $C_\Omega(f) \leq c \cdot C_{\Omega'}(f)$ UND $C_{\Omega'}(f) \leq d \cdot C_\Omega(f)$.
EINE ANALOGE AUSSAGE GILT FÜR DIE SCHALTKREISTIEFEN $D_\Omega(f)$ UND $D_{\Omega'}(f)$.

*Beweis:* Sei $S$ ein $\Omega$-Schaltkreis minimaler Größe, der $f$ berechnet. Wir können in $S$ alle Gatter (diese sind mit einer Funktion in $\Omega$ assoziiert) durch äquivalente Teilschaltkreise ersetzen, die auf der Basis $\Omega'$ beruhen (dies geht, da $\Omega'$ vollständig ist). Schlimmstenfalls erzeugt ein solcher Eingriff aus einem ursprünglichen Gatter bis zu $c$ neue Gatter; d.h. insgesamt entsteht aus einem $\Omega$-Schaltkreis $S$ der Größe $C(S)$ ein $\Omega'$-Schaltkreis $S'$ der Größe $C(S') \leq c \cdot C(S)$. Damit ist $C_\Omega(f) \leq c \cdot C_{\Omega'}(f)$ gezeigt. Analoges gilt für die Schaltkreistiefe. Die Umkehrung, von $\Omega'$ nach $\Omega$, geht entsprechend. □

Jede $n$-stellige Boolesche Funktion $f$ kann durch einen Schaltkreis, der die DNF oder KNF von $f$ realisiert, dargestellt werden, und diese besteht aus maximal $2^n$ Termen mit jeweils bis zu $n$ Variablen. Daraus ergibt sich eine allgemeine obere Schranke für die Schaltkreisgröße aller $n$-stelligen Booleschen Funktionen; und zwar besteht ein $\{\vee, \wedge, \neg\}$-Schaltkreis für einen einzelnen Term aus höchstens $2n$ Gattern; insgesamt ergibt sich $C_{\{\vee,\wedge,\neg\}}(f) \leq 2n2^n$.
Dies kann weiter verbessert werden mittels der *sel*-Funktion. Und zwar kann jede Boolesche Funktion $f$ wie folgt berechnet werden:

$$f(x_1, x_2, \ldots, x_n) = \mathit{sel}\,(x_1, f(0, x_2, \ldots, x_n), f(1, x_2, \ldots, x_n))$$

und dann rekursiv immer so weiter:




An den $2^n$ Blättern dieses Schaltkreises müssen die $2^n$ vielen Konstanten $f(0,\ldots,0,0)$, $f(0,\ldots,0,1)$, ..., $f(1,\ldots,1,1)$ eingespeist werden. Dies liefert die Schranke $C_{\{sel\}}(f) \leq 2^n$. Und als obere Schranke für die Schaltkreistiefe ergibt sich $D_{\{sel\}}(f) \leq n$.

Noch eine weitere Verbesserung erhalten wir, wenn wir die Rekursion nach $n-d$ Schritten stoppen. Dann haben wir (zunächst mal) einen Aufwand von $2^{n-d}$ Gattern. Nun müssen wir aber in die erste Schaltkreisschicht noch Funktionen der Art $f(a_1,\ldots,a_{n-d},x_{n-d+1},\ldots,x_n)$ mit $a_i \in \{0,1\}$ einspeisen. Es gibt "nur" $2^{2^d}$ viele solche Funktionen. Diese stellen wir nun alle als Schaltkreise bereit und verbinden sie entsprechend mit der untersten Schicht. Der Gatteraufwand für eine einzelne solche Funktion ist $\leq 2^d$. Insgesamt haben wir also einen Aufwand von

$$2^{n-d} + 2^{2^d} \cdot 2^d$$

Wir wählen nun (nach einigem Probieren) $d = \log(n - 2\log n)$ und erhalten:

$$\begin{aligned} C_{\{sel\}}(f) &\leq 2^{n-\log(n-2\log n)} + 2^{2^{\log(n-2\log n)}} \cdot 2^{\log(n-2\log n)} \\ &= \tfrac{2^n}{n-2\log n} + \tfrac{2^n}{n^2} \cdot (n - 2\log n) \\ &\leq \tfrac{2^n}{n/2} + \tfrac{2^n}{n} \\ &= 3 \cdot \tfrac{2^n}{n} \end{aligned}$$

Wir erhalten somit den

**Satz (Lupanov, 1958).**
ES GIBT KONSTANTEN $c$ UND $n_0$, SO DASS FÜR ALLE $n$-STELLIGEN BOOLESCHEN FUNKTIONEN $f$ UND $n \geq n_0$ GILT $C(f) \leq c \cdot 2^n/n$.

Wir haben hier die Bezugnahme auf die Basis weggelassen, da sich die $c$-Werte bei vollständigen Basen nur um einen konstanten Faktor unterscheiden.

Das heißt also: mehr als exponential in $n$ viele Gatter braucht keine Funktion zu ihrer Realisierung, wir hätten aber gerne nur polynomial-viele Gatter gehabt; aber leider können nicht alle Booleschen Funktionen – aus grundsätzlichen Gründen – durch (Größen- oder Tiefen-) "effiziente" Schaltkreise berechnet lassen. Dieses Negativ-Ergebnis ist eines der wichtigsten der Schaltkreiskomplexitätstheorie und geht auf Claude Shannon zurück.

**Satz (Shannon, 1949).**
FÜR JEDES $n$ GIBT ES $n$-STELLIGE BOOLESCHE FUNKTIONEN $f$, DIE FÜR IHRE REALISIERUNG (MITTELS NAND-GATTERN) MINDESTENS $\frac{1}{2} \cdot 2^n/n$ VIELE GATTER BENÖTIGEN. ALSO $C_{\{nand\}}(f) \geq \frac{1}{2} \cdot 2^n/n$.

*Beweis:* Sei $a(n, c)$ die Anzahl der $n$-stelligen Booleschen Funktionen, die sich mit einem Nand-Schaltkreis mit $c$ Gattern berechnen lassen. Wir schätzen diese Funktion nach oben ab: Jedes der $c$ Gatter hat 2 Eingänge; jeder dieser insgesamt $2c$ Eingänge kann mit einem der $c - 1$ anderen Gatter oder mit einem der $n$ Eingänge oder mit einer der Konstanten 0,1 verbunden sein. Dies sind pro Eingang $c + n + 1$ viele Möglichkeiten; also gibt es insgesamt nicht mehr als $(c + n + 1)^{2c}$ viele Schaltkreise der Größe $c$. (Tatsächlich ist diese Abschätzung recht grob, da wir nicht berücksichtigen, dass viele unterschiedlich verdrahtete Schaltkreise evtl. dieselbe Funktion berechnen; außerdem haben wir Schaltkreise, die unzulässig sind, da ihr Graph einen Zyklus enthält, mitgezählt). Also ist $a(n, c) \leq (c + n + 1)^{2c}$. Sei nun $c = c(n)$ minimal, so dass mit Schaltkreisen der Größe $c$ alle $n$-stelligen Booleschen Funktionen berechnet werden können. Da es $2^{2^n}$ viele verschiedene $n$-stellige Boolesche Funktionen gibt, muss für dieses $c$ gelten: $a(n, c) \geq 2^{2^n}$. Hieraus folgt mit der obigen Abschätzung: $(c + n + 1)^{2c} \geq 2^{2^n}$. Nach Logarithmieren ergibt sich

$$c \geq (1/2) \cdot 2^n / \log(c + n + 1) \geq (1/2) \cdot 2^n/n$$

Bei der letzten Ungleichung haben wir $c + n + 1 \leq 2^n$ verwendet, was sich aus dem Satz von Lupanov ergibt.

Um also *alle* $n$-stelligen Booleschen Funktionen berechnen zu können, benötigen wir Schaltkreise der Größe mindestens $(1/2) \cdot 2^n/n$. Also müssen einzelne Boolesche Funktionen existieren, die zu ihrer Berechnung mindestens diese Schaltkreisgröße benötigen. □

**Folgerung.**
ES GIBT KONSTANTEN $c$ UND $n_0$, SO DASS ES FÜR JEDES $n \geq n_0$ EINE $n$-STELLIGE BOOLESCHE FUNKTION $f$ GIBT MIT $C(f) \geq c \cdot 2^n/n$.

Tatsächlich gilt sogar, dass "die meisten" $n$-stelligen Booleschen Funktionen eine Schaltkreiskomplexität in der Größenordnung von $2^n/n$ haben (vgl. Wegener: The Complexity of Boolean Functions, Teubner-Wiley, 1987).

Ähnlich wie im letzten Satz kann man auch eine untere Schranke für die notwendige Schaltkreistiefe herleiten; und zwar gibt es $n$-stellige Boolesche Funktionen, so dass jeder Schaltkreis für die Funktion mindestens die Tiefe $n - \log\log n$ hat. (Tatsächlich ist dies wieder für "die meisten" Booleschen Funktionen der Fall).

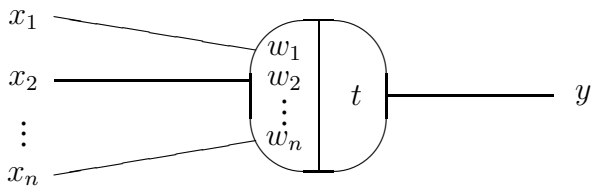
## 1.7 Perzeptrone

Wir wollen als nächstes Schaltkreise einer besonderen Art betrachten, nämlich *neuronale Netze*. Dies ist eine Sammelbezeichnung für Schaltkreise, deren Grundbaustein den menschlichen Neuronen abgeschaut ist. Das einfachste ist das *Perzeptron* (auch McCulloch-Pitts Neuron genannt): Dieses hat eine beliebige Anzahl von Eingängen $x_1, \ldots x_n$, wobei über die Eingangsleitungen Werte aus $\{0, 1\}$ fließen. Jedem Eingang $x_i$ ist ein Gewicht $w_i \in \mathbb{R}$ zugeordnet (Gewichte können auch negativ sein!). Die Funktion des Perzeptrons besteht darin, den Eingangsvektor gewichtet aufzusummieren, also $s = \sum_{i=1}^{n} x_i w_i$ zu bilden, und dann $1$ oder $0$ auszugeben, je nachdem, ob die Summe $s$ einen gewissen Schwellenwert $t \in \mathbb{R}$ erreicht (oder übersteigt) oder nicht. In Formeln ausgedrückt berechnet ein Perzeptron also die Funktion:

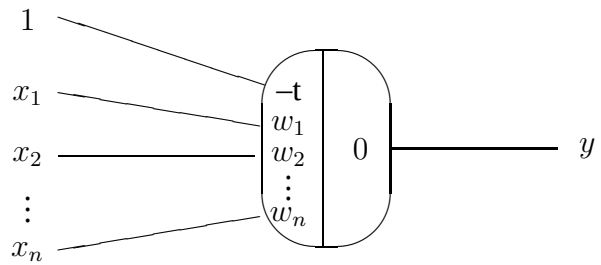
$$y = \begin{cases} 1, & \sum_{i=1}^{n} x_i w_i \geq t, \\ 0, & \sum_{i=1}^{n} x_i w_i < t \end{cases}$$

Hierbei sind die Werte $n, w_1, \ldots, w_n, t$ in dem jeweiligen Perzeptron festgelegte Parameter, die natürlich bestimmen, welche Boolesche Funktion es berechnet. Durch geringfügiges Erniedrigen des Schwellenwerts $t$ lässt sich grundsätzlich immer erreichen, dass im Falle $y = 1$ die Summe $\sum_{i=1}^{n} x_i w_i$ *echt größer* ist als der Schwellenwert $t$. Das wollen wir im Folgenden immer annehmen.

Wir verwenden folgende Symbolik:




Man kann Perzeptrone immer so normieren, dass der Schwellenwert grundsätzlich = 0 ist. Dies kann erreicht werden, indem ein weiterer Eingang hinzugenommen wird, der an die Boolesche Konstante 1 angeschlossen wird. Als Gewicht für diesen Eingang verwendet man dann $-t$:




Den Vektor $\mathbf{x} = (x_0, x_1, \ldots, x_n)$ mit $x_0 = 1$ nennt man den *erweiterten Eingabevektor*, und dementsprechend ist $\mathbf{w} = (w_0, w_1, \ldots, w_n)$ mit $w_0 = -t$ der *erweiterte Gewichtsvektor*. Mit Hilfe dieser erweiterten Vektoren lässt sich die Funktion eines Perzeptrons wesentlich einfacher beschreiben. Diese ist

$$y = \left\{ \begin{array}{ll} 1, & \sum_{i=1}^{n} w_i x_i - t > 0 \\ 0, & \text{sonst} \end{array} \right\} = \left\{ \begin{array}{ll} 1, & \mathbf{wx} > 0 \\ 0, & \text{sonst} \end{array} \right\} = [\mathbf{wx}]$$

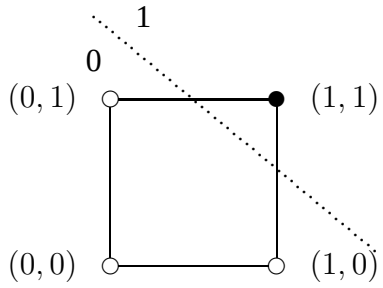
Hierbei ist $\mathbf{wx}$ das übliche Skalarprodukt der zwei (erweiterten) Vektoren. (Das Skalarprodukt von $\mathbf{a} = (a_0, \ldots, a_m)$ und $\mathbf{b} = (b_0, \ldots, b_m)$ ist definiert als $\sum_{i=0}^{m} a_i b_i$.) Die Notation $[...]$ ist so zu verstehen: Wenn das Argument größer 0 ist, so liefert dieser Ausdruck den Wert 1, sonst den Wert 0.

Die Attraktivität des Perzeptrons rührt daher, dass man sich erhofft, durch systematisches Verändern der Gewichte $w_0, w_1, \ldots, w_n$ nach gewissen "Lernregeln" das Perzeptron dazu zu bringen, eine erwünschte (aber zuvor unbekannte) Funktion zu berechnen. Dies zum Beispiel im Zusammenhang mit der Erkennung von Mustern. Hierbei kann man sich die $x_i$ als Pixel eines Rasterbildes vorstellen. Dem Perzeptron werden (zum Beispiel) Bilder von Dreiecken und Vierecken gezeigt, und immer wenn ein Bild falsch zugeordnet wird, so werden durch den "Lehrer" die Gewichte in der Weise verändert, dass das Bild korrekt erkannt wird. Auf diese Weise wird ein neuronales Netz "trainiert" (siehe nächster Abschnitt).

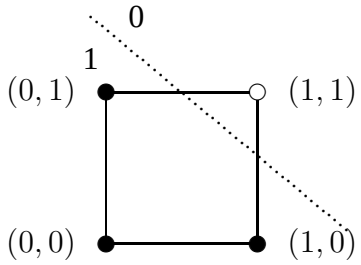
Das Modell des Perzeptrons wurde in den 50er und 60er Jahren von F. Rosenblatt eingeführt. Seine Grenzen und Möglichkeiten wurden insbesondere in dem Buch von Minsky und Papert: Perceptrons, MIT Press, 1969, eingehend untersucht.

Wir wollen uns hier zunächst nur auf die statische Betrachtung der Leistungsfähigkeit eines einzelnen Neurons konzentrieren.

Nehmen wir ein (einzelnes) Perzeptron mit 2 Eingängen. Dieses wird bestimmt durch $w_1, w_2$ und $t$. Wenn wir etwa $w_0 = -t = -1$, $w_1 = 0.6$, $w_2 = 0.6$ wählen, so ist der Funktionswert des Perzeptrons genau dann 1, wenn für $\mathbf{wx} \geq 0$, das heißt, wenn für $x_1, x_2$ gilt: $0.6x_1 + 0.6x_2 \geq 1$. Der Argumenteraum $\mathbb{B}^n$ (hier ist $n = 2$) durch die Gerade $0.6x_1 + 0.6x_2 = 1$ in zwei Hälften zerlegt:

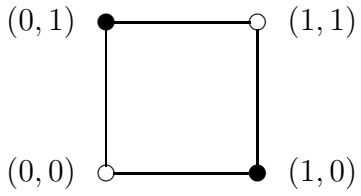



Bei dem schwarz eingezeichneten Punkt $(1,1)$, der rechts oberhalb der Trennlinie liegt, wird 1 ausgegeben. Bei den drei Punkten $(0,0)$, $(0,1)$ und $(1,0)$ wird 0 ausgegeben. Offensichtlich berechnet dieses so parametrisierte Perzeptron gerade die Und-Funktion. Deren Komplementfunktion, das Nand, erhalten wir, indem wir alle Parameter negativ nehmen: $w_0 = 1$, $w_1 = -0.6$ und $w_2 = -0.6$.




Die durch ein Perzeptron (durch geeignet festgelegte) Parameter berechenbaren Funktionen sind genau die *linear separierbaren* Funktionen: Der Argumenteraum, im allgemeinen Fall $\mathbb{B}^n$, muss durch eine "Hyperebene" der Dimension $n-1$ (im Fall $n = 2$ also eine Gerade) in zwei Hälften zerlegbar sein, so dass die eine Hälfte genau zu den Argumenten gehört, wo die Funktion den Wert 0 annimmt, die andere Hälfte zu den Argumenten, wo die Funktion den Wert 1 annimmt.

Anschaulich ist sofort klar, dass die 2-stellige Xor-Funktion nicht linear separierbar ist, und daher nicht durch ein einzelnes Perzeptron berechnet werden kann:




Das ist aber noch kein Beweis. Hier ist einer:

**Satz.**
DIE XOR-FUNKTION IST DURCH EIN EINZELNES PERZEPTRON NICHT BERECHENBAR.

*Beweis:* (nach Minsky/Papert: Perceptrons, MIT-Press, 1969) Angenommen, es gibt Koeffizienten $w_1, w_2, t \in \mathbb{R}$, so dass für alle $x_1, x_2 \in \{0, 1\}$ gilt:

$$x_1 \oplus x_2 = 1 \;\Leftrightarrow\; w_1 x_1 + w_2 x_2 - t \geq 0$$

Da $x_1 \oplus x_2 = x_2 \oplus x_1$ gilt, folgt:

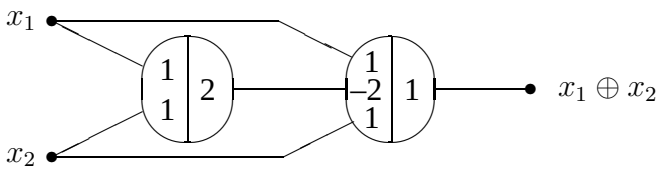
$$x_1 \oplus x_2 = 1 \;\Leftrightarrow\; w_1 x_2 + w_2 x_1 - t \geq 0$$

Indem wir die rechten Seiten der beiden obigen Charakterisierungen addieren, erhalten wir:

$$\begin{aligned} x_1 \oplus x_2 = 1 \quad &\Leftrightarrow \quad (w_1 + w_2)x_1 + (w_1 + w_2)x_2 - 2t \geq 0 \\ &\Leftrightarrow \quad (w_1 + w_2)(x_1 + x_2) - 2t \geq 0 \end{aligned}$$

Fassen wir den auf der rechten Seite vorkommenden Term $(w_1 + w_2)(x_1 + x_2) - 2t$ als Funktion von $z := x_1 + x_2$ auf, dann ist $f(z) = (w_1 + w_2)z - 2t$ eine lineare Funktion in $z$ (eine Gerade). Diese hat höchstens eine Nullstelle. Andererseits müsste sie aber mindestens 2 Nullstellen haben, denn an der Stelle $z = 0$ (entspricht $x_1 = x_2 = 0$) müsste der Wert von $f$ kleiner als Null sein; an der Stelle $z = 1$ (entspricht $x_1 = 1, x_2 = 0$) müsste der Wert von $f$ größer Null sein; und an der Stelle $z = 2$ (entspricht $x_1 = x_2 = 1$) müsste der Wert von $f$ kleiner als Null sein. Widerspruch. □

Die Xor-Funktion kann jedoch ohne weiteres durch ein mehrschichtiges Perzeptron berechnet werden. Hier ist eine Möglichkeit:

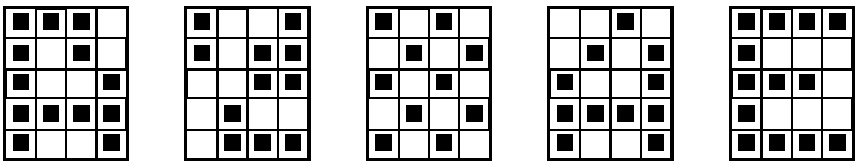



## 1.8 Lernen von Booleschen Funktionen

Nur linear separierbare Funktionen können also durch ein einzelnes Perzeptron dargestellt werden (vorausgesetzt die Parameter des Perzeptrons sind richtig eingestellt). Wir wollen nun zeigen, dass im Falle einer $n$-stelligen Booleschen Funktion $f$, die linear separierbar ist, ein Perzeptron durch einen "Lernprozess" in der Lage ist, diese Funktion nach endlicher Zeit darzustellen. Hierzu werden dem Perzeptron "Beispiele" gezeigt, dies sind (zufällig oder systematisch gewählte) $x_i \in \{0, 1\}^n$, $i = 1, 2, 3, \ldots$, und dann bei Vorliegen eines falschen Funktionswerts die Parameter des Perzeptrons systematisch gemäß einer gewissen *Lernregel* geändert. Nach einer (genügend langen) Folge von solchen Lernschritten sollte das Perzeptron dann (hoffentlich) keine Fehler mehr machen.

Das folgende Perzeptron-Konvergenztheorem besagt gerade, dass solches Lernen in der Tat nach endlich vielen Schritten erfolgreich sein muss.

*Beispiel:* Das Perzeptron soll etwa das "Konzept" des Buchstabens "A" auf einem Rasterbild erkennen. Ihm werden die folgenden Beispiele geliefert:



"A" "kein A" "kein A" "A" "kein A"

Das Perzeptron müsste hier $4 \cdot 5 = 20$ Eingänge für die Rasterpunkte haben. Nach einer genügend langen "Trainingsphase" mit Beispielen wie oben erwarten wir vom Perzeptron, dass es – vielleicht bis auf eine kleine Fehlerquote – den Buchstaben 'A' auf dem Rasterbild von 'nicht-A' unterscheidet. (Wahrscheinlich wird es so einfach nicht gehen, da die Konzepte 'A' und 'nicht-A' wohl nicht linear separierbar sind).

Es wird sich wieder als günstig erweisen, wenn wir auf den Schwellenwert 0 normieren und mit den *erweiterten* Eingabe- und Gewichtsvektoren arbeiten.

Die Lernregel können wir nun wie folgt formulieren:

Seien $Y_0$ und $Y_1$ disjunkte, und zwar linear separierbare, Mengen von "Beispielen", die für das "Training" das Perzeptrons verwendet werden sollen. Das bedeutet, dass jeder Vektor in $Y_i$ die Form $(1, y_1, \ldots, y_n)$, $y_i \in \{0, 1\}$, hat und die Wunschvorstellung ist, dass das Perzeptron schließlich einen Gewichtsvektor $\mathbf{w}$ annimmt, für den gilt:

$$\begin{array}{rcl} \mathbf{y} \in Y_1 & \Rightarrow & \mathbf{yw} > 0 \\ \mathbf{y} \in Y_0 & \Rightarrow & \mathbf{yw} < 0 \end{array}$$

In diesem Fall wäre der Lernvorgang beendet.

Das Perzeptron startet allerdings mit dem Gewichtsvektor $\mathbf{w}^0 = (0, 0, \ldots, 0)$ (oder einem anderen) und wird zunächst voraussichtlich "Fehler" machen.

Wird dem Perzeptron ein Beispiel $\mathbf{y}$ vorgelegt, und dieses wird falsch klassifiziert, so wird der Gewichtsvektor im nächsten Schritt wie folgt von $\mathbf{w}^t$ zu $\mathbf{w}^{t+1}$ geändert:

$$\mathbf{w}^{t+1} = \begin{cases} \mathbf{w}^t + \mathbf{y}, & \text{falls } \mathbf{y} \in Y_1 \text{ und } \mathbf{yw} \leq 0 \\ \mathbf{w}^t - \mathbf{y}, & \text{falls } \mathbf{y} \in Y_0 \text{ und } \mathbf{yw} \geq 0 \\ \mathbf{w}^t, & \text{sonst} \end{cases}$$

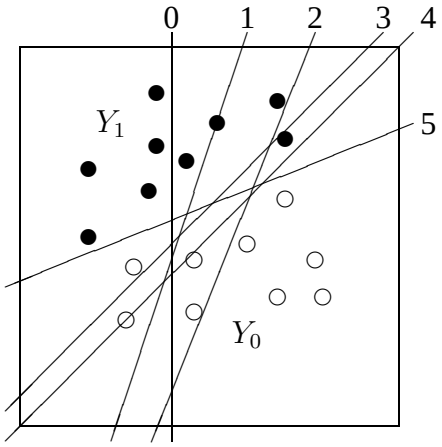
Hierbei ist die Addition bzw. Subtraktion komponentenweise zu verstehen. Die Idee ist die folgende: Wenn $\mathbf{y}$ zum Beispiel in der Menge $Y_1$ liegt, das Perzeptron also 1 ausgeben sollte, stattdessen aber 0 ausgibt (also $\mathbf{yw}^t \leq 0$), so sieht die Bilanz nach der Änderung des Gewichtsvektors wie folgt aus:

$$\mathbf{w}^{t+1}\mathbf{y} = (\mathbf{w}^t + \mathbf{y})\mathbf{y} = \mathbf{w}^t\mathbf{y} + \mathbf{yy} > \mathbf{w}^t\mathbf{y} \quad \text{da } \mathbf{y} \neq \mathbf{0}$$

Das heißt, der Wert des Skalarproduktes, der für die Entscheidung des Perzeptrons ausschlaggebend ist, wird nach oben verschoben. Es muss nicht unbedingt heißen, dass nach einem solchen Lernschritt bereits der Funktionswert des Perzeptrons korrekt =1 ist. (Es ist sogar möglich, dass bisher korrekt erkannte Eingaben durch diese Veränderung des Gewichtsvektors nun falsch klassifiziert werden).

Trotzdem, das folgende Theorem wird zeigen, dass auf lange Sicht (nach endlich vielen Lernschritten) dieses Vorgehen erfolgreich sein wird.

Man kann sich die Sachlage im zweidimensionalen Raum veranschaulichen. Die Mengen $Y_0$ und $Y_1$ sind zwei (im Prinzip) durch eine Gerade (im $n$-dimensionalen Fall: eine "Hyperebene") voneinander trennbare Punktemengen. Die Trennung, die das Perzeptron zunächst durch seinen Gewichtsvektor, zum Zeitpunkt 0, vornimmt, ist jedoch eine ganz andere. Nach Durchlaufen mehrerer Lernschritte erreicht das Perzeptron schließlich die gewünschte Trennung von $Y_0$ und $Y_1$.




Um den Beweis zu vereinfachen, ersetzen wir $Y_0$ durch $Y_0' = \{-\mathbf{y} \mid \mathbf{y} \in Y_0\}$. Die Aussage, dass $Y_0$ und $Y_1$ linear separierbar sind, kann nun so ausgedrückt werden: es gibt einen Gewichtsvektor $\mathbf{w}$ so dass $\mathbf{wy} > 0$ für alle $\mathbf{y} \in Y := Y_0' \cup Y_1$ gilt.

Wir stellen uns nun eine beliebige unendliche Folge von Elementen aus $Y$ vor, eine sog. *Trainingsfolge*, mit der Eigenschaft, dass jedes Element von $Y$ in der Folge unendlich oft auftritt. Mit dieser Folge wird nun das Perzeptron entsprechend der obigen Lernregel "trainiert". Es entsteht eine entsprechende Folge von Gewichtsvektoren. Wir bilden nun die Teilfolge $(\mathbf{w}^0, \mathbf{w}^1, \mathbf{w}^2, \ldots)$ dieser Gewichtsvektoren-Folge, so dass sich von $\mathbf{w}^j$ zu $\mathbf{w}^{j+1}$ tatsächlich eine Änderung ergibt, nämlich aufgrund eines falsch klassifizierten $\mathbf{y}^j \in Y$ in der Trainingsfolge. Das heißt, es gilt: $\mathbf{w}^{j+1} = \mathbf{w}^j + \mathbf{y}^j$ und $\mathbf{w}^j\mathbf{y}^j \leq 0$. (Man beachte, dass der Übergang von $Y_0$ zu $Y_0'$ an dieser Stelle die Formel für die Lernregel vereinfacht).

Was wir zeigen wollen ist, dass diese Folge endlich ist, das heißt, dass es einen Zeitpunkt $t$ gibt mit $\mathbf{w}^t\mathbf{y} > 0$ für alle $\mathbf{y} \in Y$. Nach dem Zeitpunkt $t$ gibt es keine Änderung des Gewichtsvektor mehr.

Da wir mit dem Nullvektor $\mathbf{w}^0$ starten, gilt also

$$\mathbf{w}^{j+1} = \mathbf{y}^1 + \mathbf{y}^2 + \ldots + \mathbf{y}^j$$

Wir müssen beweisen, dass $j$ nicht beliebig groß werden kann.

Sei nun $\mathbf{w}$ ein beliebiger Lösungsvektor, das heißt, es gilt $\mathbf{yw} > 0$ für alle $\mathbf{y} \in Y$. Sei $\alpha$ eine kleine positive Konstante, so dass $\mathbf{yw} \geq \alpha$ für alle $\mathbf{y} \in Y$ gilt.

Nun können wir abschätzen:

$$\mathbf{w}^{j+1}\mathbf{w} \;=\; (\mathbf{y}^1 + \ldots + \mathbf{y}^j)\mathbf{w} \;=\; \mathbf{y}^1\mathbf{w} + \ldots + \mathbf{y}^j\mathbf{w} \;\geq\; j\alpha$$

Mit der Cauchy-Schwarz-Ungleichung, die besagt, dass immer $(\mathbf{ab})^2 \leq (\mathbf{aa}) \cdot (\mathbf{bb})$ gilt, folgern wir:

$$j^2\alpha^2 \;\leq\; (\mathbf{w}^{j+1}\mathbf{w})^2 \;\leq\; (\mathbf{w}^{j+1}\mathbf{w}^{j+1}) \cdot (\mathbf{ww})$$

Dies ergibt:

$$\mathbf{w}^{j+1}\mathbf{w}^{j+1} \geq j^2\alpha^2/(\mathbf{ww})$$

Das heißt, dass der Wert des Ausdrucks $\mathbf{w}^{j+1}\mathbf{w}^{j+1}$ quadratisch in $j$ anwächst. Wir werden zeigen, dass solch ein quadratisches Wachstum nicht unendlich oft möglich ist, denn es gilt mittels $\mathbf{w}^{j+1} = \mathbf{w}^j + \mathbf{y}^j$ und $\mathbf{w}^j\mathbf{y}^j \leq 0$ die Abschätzung:

$$\mathbf{w}^{j+1}\mathbf{w}^{j+1} = \mathbf{w}^j\mathbf{w}^j + 2 \cdot (\mathbf{w}^j\mathbf{y}^j) + \mathbf{y}^j\mathbf{y}^j \leq \mathbf{w}^j\mathbf{w}^j + \mathbf{y}^j\mathbf{y}^j$$

Dies ergibt, nach $j$-maliger Anwendung:

$$\mathbf{w}^{j+1}\mathbf{w}^{j+1} \;\leq\; \mathbf{y}^1\mathbf{y}^1 + \cdots + \mathbf{y}^j\mathbf{y}^j \;\leq\; j \cdot \max\{\mathbf{yy} \mid \mathbf{y} \in Y\} \leq j \cdot (n+1)$$

Indem wir diese Ungleichungen zusammenfügen, erhalten wir:

$$j^2\alpha^2/(\mathbf{ww}) \;\leq\; \mathbf{w}^{j+1}\mathbf{w}^{j+1} \;\leq j \cdot (n+1)$$

also

$$j \leq (n+1) \cdot \mathbf{ww}/\alpha^2$$

Das heißt, $j$ ist durch eine Konstante, die nur von der Beispielmenge, der Dimension $n$, und dem gewählten Lösungsvektor $\mathbf{w}$ abhängt, nach oben beschränkt. Mit anderen Worten, es kann nur endlich viele echte Lernschritte geben, bis sich der Gewichtsvektor des Perzeptrons nicht mehr ändert. (Diese Anzahl der Lernschritte allerdings konkret auszurechnen, erweist sich als schwierig.) Dass dieser Gewichtsvektor tatsächlich ein Lösungsvektor ist, ergibt sich daraus, dass jedes Beispiel in der Trainingsfolge nach Voraussetzung unendlich oft auftritt. Das heißt, wenn nicht alle Beispiele durch den erreichten Gewichtsvektor korrekt klassifiziert würden, müsste mindestens ein weiterer Änderungsschritt ausgelöst werden.

Was wir bewiesen haben, ist das berühmte Perzeptron-Konvergenztheorem:

**Satz.**
GEGEBEN SEIEN ZWEI LINEAR SEPARIERBARE MENGEN VON PUNKTEN $Y_0, Y_1$ UND EINE UNENDLICHE TRAININGSFOLGE, WIE OBEN BESCHRIEBEN. DANN ERREICHT DAS PERZEPTRON NACH ENDLICH VIELEN LERNSCHRITTEN (GEMÄSS DER OBEN BESCHRIEBENEN LERNREGEL) EINEN GEWICHTSVEKTOR, SO DASS DAS PERZEPTRON ALLEN PUNKTEN IN $Y_0$ DEN WERT 0, UND ALLEN PUNKTEN IN $Y_1$ DEN WERT 1 ZUWEIST.

## 1.9 Codes

Bei der Speicherung und Übertragung von Daten tritt das Problem auf, dass diese Daten in irgendeiner Form codiert werden müssen. Ferner sollen die Daten gegen (seltene, aber doch nicht zu ignorierende) zufällige Störungen gesichert werden. In der Codierungstheorie beschäftigt man sich mit der Frage, inwieweit hierfür *Codes* entworfen werden können, die die Möglichkeit geben, aufgetretene Fehler zu erkennen, und womöglich sogar zu korrigieren. Wir wollen hier nur die absoluten Grundlagen dieser Theorie besprechen.

*Beispiel:* Wird etwa die Nachricht "Meine Telefonnummer ist 3615" an einigen Stellen fehlerhaft übertragen, so könnte beim Empfänger etwa die Nachricht "Mexne Teleuonnummer isd 3415" ankommen. Der Empfänger weiß dennoch, was die ersten drei Wörter bedeuten, denn die Sprache ist *redundant*, d.h. enthält mehr Informationen als zum Verständnis notwendig sind. Mit der Telefonnummer verhält es sich anders: Aus der empfangenen Nachricht ist nicht einmal zu erkennen, dass ein Fehler aufgetreten ist.

In der Codierungstheorie wird studiert, wie man Nachrichten möglichst "günstig" redundant machen kann, wobei "günstig" verschiedenes bedeuten kann: aufgetretene Fehler sollten (möglichst algorithmisch einfach) erkannt werden und darüberhinaus sollte es (bei wenigen Fehlern) möglich sein, die ursprüngliche Nachricht (algorithmisch einfach) zu rekonstruieren.

*Beispiel:* Der Buchhandel verwendet zur Identifizierung von Büchern die *International Standard Book Number* (ISBN). Ein für Informatik-Studenten nützliches Buch trägt zum Beispiel die Nummer

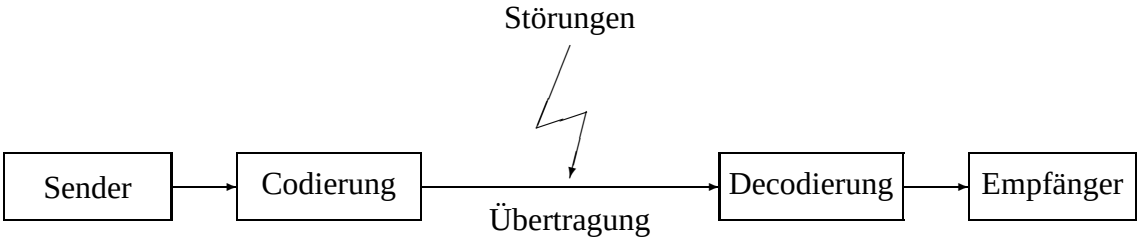
3-411-15641-4

Die erste Ziffer bezeichnet das Erscheinungsland (3=Deutschland), die nächsten 3 Ziffern den Verlag und die anschließenden 5 Ziffern das Buch. Bei der letzten Ziffer handelt sich um eine sog *Prüfziffer*. Schreibt man die Buchnummer als Tupel $b_1 b_2 \ldots b_9 b_{10}$, dann berechnet sich die Prüfziffer als

$$b_{10} = (1 \cdot b_1 + 2 \cdot b_2 + \ldots + 9 \cdot b_9) \text{ MOD } 11$$

wobei anstelle von $b_{10} = 10$ jedoch $b_{10} = X$ geschrieben wird. Sollte an einer der 10 Stellen ein Fehler aufgetreten sein, so kann diese Tatsache erkannt werden. Auch mehrfache Fehler können oft erkannt werden. Eine automatische Fehlerkorrektur ist allerdings nicht möglich; die Nummer muss gegebenenfalls neu eingegeben oder übertragen werden.

Die Datenübertragung mit Hilfe von Codes kann man sich schematisch wie folgt vorstellen:




Verwenden wir beispielsweise für die beiden möglichen Nachrichten $A$ und $B$ die Codes 00000 und 11111, so könnte sich etwa folgende Situation ergeben:

| Nachricht | $\rightarrow$ | Codewort | $\rightarrow$ | gestörtes Codewort | $\rightarrow$ | vermutetes Codewort | $\rightarrow$ | vermutete Nachricht |
|---|---|---|---|---|---|---|---|---|
| $A$ | $\rightarrow$ | 00000 | $\rightarrow$ | 01001 | $\rightarrow$ | 00000 | $\rightarrow$ | $A$ |

Man geht davon aus, dass Übertragungsfehler relativ selten sind. Daher ist die folgende "Maximum-Likelyhood-Decodierung" sinnvoll: Man decodiere das empfangene Tupel $v = v_1 v_2 \dots v_n$ als ein Codewort $c = c_1 c_2 \dots c_n$, das sich von $v$ an möglichst wenigen Stellen unterscheidet.

Der *(Hamming-)Abstand* $d(x, y)$ von zwei Vektoren $x, y \in \mathbb{B}^n$ ist die Anzahl der Stellen, an denen sich $x$ und $y$ unterscheiden. Das *(Hamming-)Gewicht* $w(x)$ eines Vektors $x \in \mathbb{B}^n$ ist definiert als die Anzahl der von 0 verschiedenen Stellen von $x$.

Es gilt offensichtlich $w(x) = d(x, 00 \dots 0)$ und $d(x, y) = w(x \oplus y)$.

Ein *Code* ist eine nicht-leere, endliche Teilmenge von $\mathbb{B}^+ = \mathbb{B} \cup \mathbb{B}^2 \cup \mathbb{B}^3 \cup \dots$

Falls die Elemente eines Codes $C$ (die *Codewörter* genannt werden) alle dieselbe Länge haben, also $C \subseteq \mathbb{B}^n$ für ein $n \in \mathbb{N}$, so spricht man von einem *Blockcode* (der Länge $n$) Diese Länge $n$ nennt man auch die *Dimension* des Codes. Im folgenden interessieren wir uns nur für Blockcodes.

Der *Minimalabstand* eines Codes $C$ ist definiert als

$$d(C) \;=\; \min\{d(x, y) \mid x, y \in C,\, x \neq y\}$$

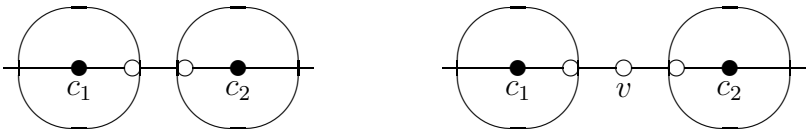
Anschaulich bezeichnen wir alle Wörter mit Hamming-Distanz $\leq t$ von einem Codewort $c$ als die *Kugel* um $c$ vom Radius $t$.

Ein Code heißt *$t$-Fehler-korrigierend*, falls für alle $c \in C$ und alle $c'$ mit $d(c, c') \leq t$ gilt, dass es kein anderes Codewort (als $c$) mit Abstand $\leq t$ zu $c'$ gibt. In anderen Worten: Nach dem Maximum-Likelyhood-Prinzip kann $c'$ eindeutig dem Codewort $c$ zugeordnet werden.

Der folgende Satz ist aufgrund dieser Definitionen offensichtlich:

**Satz.**
JEDER CODE MIT MINIMALABSTAND $d$ IST $\lfloor (d-1)/2 \rfloor$-FEHLER-KORRIGIEREND.

Die folgende Skizze zeigt einen Code mit Minimalabstand 3 und einen mit Minimalabstand 4; beide sind 1-Fehler-korrigierend. (Es gilt $\lfloor (3-1)/2 \rfloor = \lfloor (4-1)/2 \rfloor = 1$). Bei einem verfälschten Codewort, das in der Kugel vom Radius 1 um ein korrektes Codewort (wie $c_1, c_2$) liegt, kann auf das entsprechende Codewort rückgeschlossen werden. Im Falle des Wortes $v$ ist zwar das Vorliegens eines Fehlers feststellbar, dieser kann aber nicht eindeutig korrigiert werden.




Ein Blockcode $C \subseteq \mathbb{B}^n$ heißt *systematisch* in den Stellen $i_1, \ldots, i_k$, wenn es zu jedem Vektor $u = (u_1, \ldots, u_k) \in \mathbb{B}^k$ *genau ein* Codewort $c = (c_1, \ldots, c_n)$ gibt mit $u_1 = c_{i_1}, \ldots, u_k = c_{i_k}$. Man nennt $u$ dann die *Nachricht*, die als Codewort $c$ codiert wird.

Unter einem $[n, k]$ Code versteht man einen in $k$ Stellen systematischen Code der Länge $n$. Ein $[n, k, d]$ Code ist ein $[n, k]$ Code mit Minimalabstand $d$.

*Beispiel:* Ein häufig vorkommender Blockcode besteht aus 8 Binärstellen (einem Byte), wobei das 8. Bit als Parity-Bit verwendet wird und nur die ersten 7 Bits informationstragend sind. Das 8. Bit berechnet sich zu $b_1 \oplus \ldots \oplus b_7$ aus den ersten 7 Bits. Dies ist ein $[8, 7, 2]$ Code, denn der Minimalabstand ist 2. Es können zwar 1-Bit-Fehler als solche erkannt werden, aber eine Fehlerkorrektur ist nicht möglich, denn es gilt $\lfloor (2-1)/2 \rfloor = 0$.

*Beispiel:* Der oben angegebene ISBN-Code ist ein [10,9,2] Code, allerdings nicht über dem 2-elementigen Körper $\mathbb{B}$ (wie in der Definition vorgesehen), sondern über $\mathbb{Z}_{11}$.

In den meisten Anwendungen kommen *lineare Codes* vor; und zwar heißt ein Code linear, falls mit je zwei Codewörtern $x, y$ immer auch $x \oplus y$ (bitweises Xor) Codewort ist. Alle bisher betrachteten Codes sind linear.

In einem Linearcode lässt sich der Minimalabstand besonders einfach bestimmen:

$$d \;=\; \min\{w(x) \mid x \in C,\, x \neq 00\ldots0\}$$

Bei linearen Codes kann aus der jeweiligen Nachricht mit Mitteln der linearen Algebra (Matrizenmultiplikation) das zugeordnete Codewort bestimmt werden. In ähnlicher Weise ist eine Decodierung mit Fehlerkorrektur mit geeigneten Matrizenoperationen möglich (siehe Spezialliteratur bzw. Spezialvorlesungen).

Bei der Konstruktion eines Codes $C$ sind zwei Ziele gegenläufig:

1. Der Minimalabstand $d(C)$ sollte möglichst groß sein (= gute Fehlerkorrektur),
2. $|C|$ sollte möglichst groß sein (= viele Nachrichten codierbar).

Bei einem $t$-Fehler-korrigierenden Code $C \subseteq \mathbb{B}^n$ befinden sich in jeder Kugel um ein Codewort vom Radius $t$ genau

$$\binom{n}{0} + \binom{n}{1} + \ldots + \binom{n}{t}$$

viele Wörter. Insgesamt stehen aber nur $|\mathbb{B}^n| = 2^n$ viele Wörter zur Verfügung. Damit erhalten wir sofort den folgenden Satz.

**Satz (Hammingschranke).**
FÜR DIE ANZAHL DER CODEWÖRTER IN EINEM $t$-FEHLER-KORRIGIERENDEN CODE $C$ DER LÄNGE $n$ GILT:

$$|C| \leq 2^n / \left( \binom{n}{0} + \binom{n}{1} + \ldots + \binom{n}{t} \right)$$

Wenn bei einem Code hier sogar die Gleichheit gilt, so sprechen wir von einem *$t$-perfekten* Code. Im Falle eines perfekten Codes wird der Raum $\mathbb{B}^n$ durch die Kugeln vom Radius $t$ um die Codewörter lückenlos ausgefüllt.

*Beispiel:* Für den obigen Code $C = \{00000, 11111\}$ ist $n = 5$ und $t = 2$. Wegen $2^5/(1 + 5 + 10) = 2$ ist dieser Code perfekt.

Perfekte Codes (mit mehr als 2 Codewörtern) sind nicht ganz einfach zu konstruieren. Wir verweisen auf die Spezialliteratur.

# Teil 2

# Binäre Relationen und Graphen

## 2.1 Grundbegriffe

Begriffe aus der Theorie der Relationen durchdringen viele Bereiche der Informatik (Beispiel: relationale Datenbanken). Daher sollen hier die wichtigsten zusammengefasst werden.

Seien $A$ und $B$ zwei Mengen. Das *kartesische Produkt* von beiden ist

$$A \times B = \{(a, b) \mid a \in A,\, b \in B\}$$

Eine Menge $R \subseteq A \times B$ heißt (zweistellige) *Relation*. Anstatt "$(a, b) \in R$" schreibt man oft auch "$a\, R\, b$".

Falls eine Relation $R$ die Eigenschaft hat, dass es für jedes $a$ genau ein $b$ gibt mit $a\, R\, b$ so heißt $R$ eine *Funktion* (welche wir meist mit kleinen Buchstaben $f, g, h, \ldots$ bezeichnen). Sei $f$ eine Funktion. Anstelle von $f \subseteq A \times B$ schreiben wir $f : A \longrightarrow B$. Mit $f(x)$ bezeichnen wir das mit $x$ in Relation stehende Element $y$. Wenn für alle $x, x'$ mit $x \neq x'$ gilt $f(x) \neq f(x')$, so heißt $f$ *injektiv*. Falls es für alle $y \in B$ ein $x \in A$ mit $f(x) = y$ gibt, so heißt $f$ *surjektiv*. Eine injektive und surjektive Funktion heißt *bijektiv*. Mit $f^{-1}(y)$ bezeichnen wir die Menge aller $x$ mit $f(x) = y$. In dieser Terminologie bedeutet "$f$ ist injektiv" gerade $|f^{-1}(y)| \leq 1$; und "$f$ ist surjektiv" gerade $|f^{-1}(y)| \geq 1$ (für alle $y$).

Existiert eine bijektive Funktion zwischen zwei Mengen $A$ und $B$, so heißen $A$ und $B$ *gleichmächtig* (oder von gleicher Kardinalität). Wenn eine Menge gleichmächtig, wie $\mathbb{N}$ ist, so heißt sie *abzählbar unendlich*. Eine endliche oder abzählbar unendliche Menge heißt (höchstens) *abzählbar*.

Die *Potenzmenge* einer Menge $M$ ist die Menge aller ihrer Teilmengen, mit $\mathcal{P}(M)$ oder $2^M$ bezeichnet. Also:

$$2^M = \{A \mid A \subseteq M\}$$

Falls $|M| = k$, so ist $|2^M| = 2^k$. (Begründung: Für jedes der $k$ Elemente von $M$ gibt es die beiden Möglichkeiten, das Element in eine Teilmenge aufzunehmen, oder auch nicht.

Deshalb gibt es genau $\underbrace{2 \cdot 2 \cdot \ldots \cdot 2}_{k} = 2^k$ viele Möglichkeiten, eine Teilmenge von $M$ zu konstruieren.)

Falls $M$ abzählbar unendlich ist, so ist $2^M$ nicht abzählbar (auch als *überabzählbar* bezeichnet). (Begründung: Da $M$ abzählbar unendlich ist, lässt sich $M$ schreiben als

$$M = \{a_0, a_1, a_2, \ldots\}$$

Das heißt, es gibt eine Indizierung der Elemente von $M$ mit Zahlen aus $\mathbb{N}$. Wenn $2^M$ abzählbar sein sollte, so gibt es ebenfalls eine solche, fiktive Indizierung:

$$2^M = \{M_0, M_1, M_2, \ldots\}$$

Definiere nun die Menge $D = \{a_i \mid a_i \notin M_i\}$. Diese Menge $D$ ist eine Teilmenge von $M$, also ein Element der Potenzmenge von $M$. Daher muss diese einen Index, sagen wir $j$, haben, also $D = M_j$. Dies ergibt aber der Widerspruch $j \in D \Leftrightarrow j \notin M_j$.)

Sind $R, S$ zwei Relationen, so ist die *Komposition* von $R$ und $S$ definiert als

$$R \circ S = \{(a, c) \mid \exists b : \; a\, R\, b,\; b\, S\, c\}$$

Mit $R^2$ bezeichnen wir die Relation $R \circ R$.

Die Relation

$$R^T = \{(b, a) \mid (a, b) \in R\}$$

heißt die zu $R$ *transponierte* (oder inverse) Relation.

Es gilt:

$$\begin{aligned} (R \circ S) \circ T &= R \circ (S \circ T) \\ (R \circ S)^T &= S^T \circ R^T \end{aligned}$$

Die wichtigsten Eigenschaften von Relationen:

| | |
|---|---|
| **reflexiv**: | $x\, R\, x$ |
| **symmetrisch**: | $x\, R\, y \;\Rightarrow\; y\, R\, x$ |
| **transitiv**: | $x\, R\, y,\; y\, R\, z \;\Rightarrow\; x\, R\, z$ |
| **antisymmetrisch**: | $x\, R\, y,\; y\, R\, x \;\Rightarrow\; x = y$ |
| **konnex**: | $x\, R\, y \;\vee\; y\, R\, x$ |
| **irreflexiv**: | $\neg(x\, R\, x)$ |

Hierbei sollen alle Ausdrücke *für alle* $x, y, z$ aus der betreffenden Grundmenge $X$ gelten.

Beachte: “antisymmetrisch” ist nicht dasselbe wie “nicht symmetrisch”; “irreflexiv” ist nicht dasselbe wie “nicht reflexiv”.

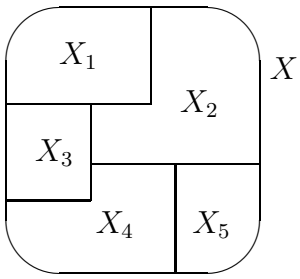
## 2.2 Äquivalenzrelationen

Eine Relation, die reflexiv, symmetrisch und transitiv ist, heißt *Äquivalenzrelation*. Ist eine Äquivalenzrelation $R$ auf einer Menge $X$ gegeben, so wird $X$ in natürlicher Weise in disjunkte Teilmengen zerlegt, d.h. es gibt (endlich oder unendlich viele) Mengen

$X_1, X_2, \ldots$ mit:

$$X = \bigcup_i X_i \quad \text{und} \quad X_i \cap X_j = \emptyset \text{ für } i \neq j$$

*Skizze:*




Und zwar umfasst eine Teilmenge $X_i$ nur zueinander in Relation stehende Elemente, während kein Element von $X_j$, $i \neq j$, mit einem Element in $X_i$ in Relation steht. Indem man aus jeder der Mengen $X_i$ ein beliebiges Element $x_i$ auswählt, kann man auch schreiben:

$$X_i = \{x \mid x\, R\, x_i\} = \{x \mid x_i\, R\, x\} =: [x_i]_R$$

Die Mengen $X_i = [x_i]_R$ heißen *Äquivalenzklassen*. Mit *Index*$(R)$ bezeichnen wir die Anzahl der durch $R$ erzeugten Äquivalenzklassen, d.h. *Index*$(R)$ ist die maximale Anzahl von Elementen $x_i$, so dass diese paarweise nicht in Relation stehen. *Index*$(R)$ kann endlich oder unendlich sein. Jedes Element $x$ kann zur Bezeichnung seiner Äquivalenzklasse, nämlich $[x]_R$, herangezogen werden; $x$ heißt dann *Repräsentant* der Äquivalenzklasse. (Wenn klar ist, welches die zugrunde liegende Äquivalenzrelation $R$ ist, so schreiben wir auch einfach $[x]$ statt $[x]_R$).

*Beispiel:* Sei auf der Grundmenge $\mathbb{N} = \{0, 1, 2, \ldots\}$ die Relation $R$ definiert als

$$m\, R\, n \quad \text{gdw} \quad m \bmod 5 = n \bmod 5$$

also wenn die Zahlen $m$ und $n$ bei Division durch 5 denselben Rest ergeben (anders ausgedrückt: $m \equiv n \pmod 5$). In diesem Fall ist *Index*$(R) = 5$, und zwar ist

$$\begin{aligned}
[0]_R &= \{0, 5, 10, \ldots\} \\
[1]_R &= \{1, 6, 11, \ldots\} \\
[2]_R &= \{2, 7, 12, \ldots\} \\
[3]_R &= \{3, 8, 13, \ldots\} \\
[4]_R &= \{4, 9, 14, \ldots\}
\end{aligned}$$

Mit $X/R$ bezeichnen wir das Mengensystem $\{[x]_R \mid x \in X\}$. Dieses wird als *Quotientenmenge* (oder auch *Faktormenge*) von $X$ nach $R$ bezeichnet. Im obigen Beispiel bezeichnet man $\mathbb{Z}/R = \{[0], [1], [2], [3], [4]\}$ oft mit $\mathbb{Z}_5$.

Im folgenden werden wir oft von der Äquivalenz von Formeln, Automaten, Maschinen oder Grammatiken reden. Diese Sprechweise bedeutet immer, dass das zueinander in Relation stehen dieser Objekte behauptet wird – in Bezug auf eine geeignete Äquivalenzrelation.

## 2.3 Halbordnungen

Weitere oft vorkommende Relationen sind *Halbordnungen* (oder partielle Ordnungen). Eine Relation heißt Halbordnung, falls sie reflexiv, antisymmetrisch und transitiv ist. Von der Bezeichnung her verwendet man oft das Zeichen $\leq$. Sei $X$ die Grundmenge. Dann heißt das Paar $(X, \leq)$ eine partiell geordnete Menge (auch: *poset = partially ordered set*).

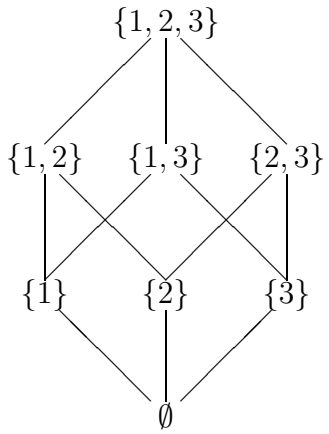
Eine Halbordnung heißt *Totalordnung* (oder lineare Ordnung), falls sie zusätzlich noch konnex ist, d.h. wenn es keine bzgl. $\leq$ unvergleichbaren Elemente gibt.

*Bemerkung:* Machmal wird in der Literatur bei der Definition einer Halbordnung auf die Reflexivität verzichtet. Bei Vorhandensein der Reflexivität wird dies dann besonders betont: “reflexive Halbordnung”.

Halbordnungen auf einer endlichen Grundmenge können anschaulich als *Hasse-Diagramm* gezeichnet werden. Und zwar wird $y$ in der Ebene oberhalb von $x$ gezeichnet, wenn $x \leq y$ gilt. Es wird nicht für jede bestehende Relation $x \leq y$ eine Kante gezeichnet, sondern nur diejenigen, die sich nicht durch Transitivität und Reflexivität “automatisch” ergeben.

*Beispiel:* Grundmenge sei die Potenzmenge (die Menge aller Teilmengen) von $\{1, 2, 3\}$. Zwei solche Teilmengen seien in Relation genau dann, wenn die erste in der zweiten enthalten ist. Man prüft leicht nach, dass es sich bei dieser Relation tatsächlich um eine Halbordnung handelt.

Das zugehörige Hasse-Diagramm ist:



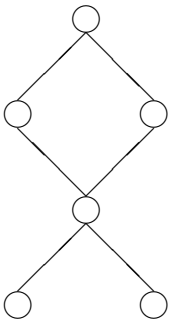

Das Hasse-Diagramm einer Totalordnung stellt immer eine lineare Kette dar:

Sei $(X, \leq)$ eine partiell geordnete Menge. Dann heißt $x \in X$

> **kleinstes Element (bzw. größtes Element),** falls für alle $y \in X$ gilt $x \leq y$ (bzw. $y \leq x$). (Dies heißt insbesondere, dass $x$ mit allen Elementen von $X$ vergleichbar sein muss).
>
> **minimales Element (bzw. maximales Element),** falls es kein $y \in X$ gibt mit $y < x$, d.h. $y \leq x$ und $y \neq x$ (bzw. $x < y$).

Falls ein größtes (bzw. kleinstes) Element existiert, so ist dieses eindeutig bestimmt. Minimale (bzw. maximale) Elemente kann es durchaus mehrere geben.

*Beispiel:* In der folgenden (durch ein Hasse-Diagramm gegebenen) Halbordnung gibt es ein größtes (und damit auch maximales) Element und 2 minimale Elemente, aber kein kleinstes Element.



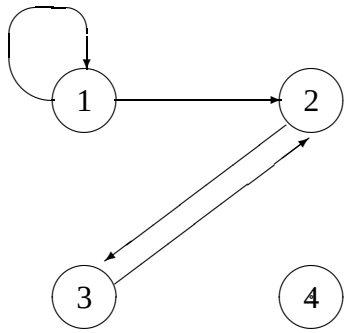
## 2.4 Graphen

Das theoretische Konzept eines Graphen durchdringt alle Bereiche der Informatik. Systeme, die aus Objekten und Beziehungen zwischen diesen Objekten bestehen, werden oft als Graphen modelliert. Daher sollen hier die wichtigsten Begriffe aus der Graphentheorie hier zusammengefasst werden.

Graphen sind zunächst einmal nichts anderes als eine graphische Veranschaulichung von endlichen Relationen; und zwar wird für jedes Element der Grundmenge $X$ ein Punkt (ein

sog. *Knoten*) gezeichnet; und für jedes Element $(x, y)$ der Relation wird ein Pfeil (eine sog. *Kante*) von $x$ nach $y$ gezeichnet.

*Beispiel:* Der zu der Relation $R = \{(1, 1), (1, 2), (2, 3), (3, 2)\}$ auf der Grundmenge $X = \{1, 2, 3, 4\}$ gehörige Graph ist:




Formal definieren wir einen *gerichteten* Graphen $G$ als ein Paar $G = (V, E)$, wobei $V$ die (endliche) Menge der Knoten (engl.: vertices) und $E \subseteq V \times V$ die Menge der Kanten (engl.: edges) ist. Die Bezeichnung „gerichtet“ rührt daher, dass die Kanten eine Richtung (eine Pfeilspitze) haben. Man bezeichnet solche Graphen oft auch als *Digraphen* (von *directed graph*).

Im Unterschied hierzu betrachten wir auch *ungerichtete* Graphen $G = (V, E)$. Hier ist wieder $V$ die Menge der Knoten und $E \subseteq \binom{V}{2}$ die Menge der Kanten. Hierbei ist $\binom{V}{2}$ die Menge aller zweielementigen Teilmengen von $V$.

Bei ungerichteten Graphen sind üblicherweise *Schlingen* nicht zugelassen (also Kanten, die von $x$ nach $x$ führen). Dies wird hier durch die formale Definition bereits ausgeschlossen, denn $\{x, x\}$ ist keine zweielementige Menge. (Graphen ohne Schlingen und Mehrfachkanten werden manchmal *schlichte Graphen* genannt).

Sollen Schlingen in einer bestimmten Anwendung zugelassen sein, so muss dies explizit gesagt werden. Ähnlich verhält es sich mit den sog. *Mehrfachkanten*: Dies sind mehrere Kanten zwischen einem Knoten $x$ und einem Knoten $y$. Will man Schlingen und Mehrfachkanten zulassen und mathematisch modellieren, so muss man statt (Knoten- und Kanten-) *Mengen* jetzt *Multimengen* verwenden. Hier dürfen Objekte in mehrfacher Ausfertigung vorkommen und verschmelzen nicht wie bei einer Menge zu einem einzelnen Element.

Sind zwei Knoten mit einer (gerichteten oder ungerichteten) Kante verbunden, so heißen sie *Nachbarn*. Bei gerichteten Graphen kann man darüber hinaus unter den Nachbarn zwischen den *Vorgängern* und den *Nachfolgern* unterscheiden.

Der *Grad* eines Knotens $v \in V$ ist die Anzahl seiner mit ihm verbundenen Kanten, welcher mit $d(v)$ bezeichnet wird. Bei gerichteten Graphen kann man darüber hinaus zwischen dem *Eingangsgrad* (die Anzahl der in $v$ hineinführenden Kanten, bezeichnet mit

$d_{in}(v)$) und dem *Ausgangsgrad* (die Anzahl der aus $v$ herausführenden Kanten, bezeichnet mit $d_{out}(v)$) unterscheiden. Es gilt: $d(v) = d_{in}(v) + d_{out}(v)$.

**Satz.**

$$\sum_{v \in V} d(v) = 2 \cdot |E|.$$

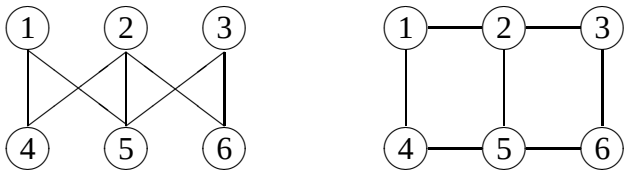
Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ heißen *isomorph*, wenn sie sich höchstens in der Bezeichnung ihrer Knoten unterscheiden, ansonsten aber dieselbe Struktur haben. Das heißt, es muss eine bijektive Funktion $f : V_1 \longrightarrow V_2$ existieren mit der Eigenschaft:

$$(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$$

(Diese Definition ist für gerichtete Graphen formuliert, gilt jedoch auch sinngemäß für ungerichtete Graphen).

Wir schreiben auch $f(G_1) = G_2$. Sei $G = (V, E)$ ein Graph und $f$ eine Bijektion (Permutation) auf der Menge $V$. Dann heißt $f$ ein *Automorphismus*, falls $f(G) = G$. Die identische Abbildung stellt immer einen Automorphismus dar. Ein Graph, der keinen anderen Automorphismus besitzt außer der identischen Abbildung heißt *rigide*. Mit $aut(G)$ bezeichnen wir die Menge der Automorphismen von $G$. Diese Menge, zusammen mit der Funktionen-Komposition, bildet eine Gruppe (mit der identischen Abbildung als neutrales Element).

*Beispiel:* Die folgenden beiden Graphen $G_1$ und $G_2$ sind isomorph:




$G_1$ $G_2$

Ein möglicher Isomorphismus (d.h. die gesuchte Funktion $f$) ist $\binom{1\ 2\ 3\ 4\ 5\ 6}{1\ 5\ 3\ 4\ 2\ 6}$. Insgesamt gibt es 4 Isomorphismen zwischen $G_1$ und $G_2$. Somit ist $|aut(G_1)| = |aut(G_2)| = 4$.

Wenn wir die vorkommenden Knotengrade eines Graphen der Größe nach absteigend sortieren, erhalten wir die *Gradsequenz* des Graphen. Im obigen Beispiel haben beide Graphen die Gradsequenz $(3, 3, 2, 2, 2, 2)$. dass die Gradsequenzen zweier isomorpher Graphen übereinstimmen müssen, ist klar. Aber leider ist diese Bedingung zur Charakterisierung der Isomorphie nicht hinreichend. (Dann wäre das nachfolgende Graphenisomorphieproblem sehr einfach zu lösen).

Man beachte, dass nicht jede absteigend sortierte Folge von natürlichen Zahlen eine Gradsequenz ist; etwa $(1, 1, 1)$. Um festzustellen, ob eine absteigend sortierte Zahlenfolge eine

Gradsequenz darstellt, kann man diese Folge solange nach einem bestimmten Schema vereinfachen, bis ggf. die leere Folge entsteht. Genau in diesem Fall stellt die Ausgangsfolge eine Gradsequenz dar. Ein Reduktionsschritt wird folgendermaßen vollzogen: Man streicht die erste, also die größte Zahl, sagen wir $k$, aus der Folge und zieht von den nachfolgenden $k$ Zahlen 1 ab. Danach sortiert man die so entstandene Zahlenfolge erst wieder, bevor man den nächsten Reduktionsschritt anwendet.

Das *Graphenisomorphieproblem* ist die algorithmische Aufgabe, möglichst effizient festzustellen, ob zwei gegebene Graphen isomorph sind oder nicht. Es ist nicht bekannt, ob es für dieses Problem effiziente Algorithmen gibt, alle bisherigen Algorithmen sind exponentiell – außer für bestimmte Spezialfälle, wie zum Beispiel Bäume. Das Problem hat deshalb eine so große Bedeutung, da sich das Isomorphieproblem für viele andere algebraische (oder Informatik-) Objekte als ein Graphenisomorphieproblem auffassen lässt (zum Beispiel: Automaten, Gruppen, Halbgruppen). Denselben Status wie das Graphisomorphieproblem hat das *Graphautomorphieproblem*: gegeben ein Graph $G$, stelle fest, ob dieser einen Automorphismus $f \neq id$ besitzt, also ob $|aut(G)| > 1$.

### 2.4.1 Wege und Kreise in Graphen

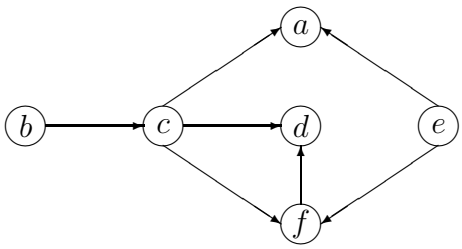
Ein *Weg* (oder Pfad) in einem (gerichteten oder ungerichteten) Graphen $G = (V, E)$ ist eine Folge von Knoten $(v_0, v_1, \ldots, v_k)$ aus $V$, so dass $(v_{i-1}, v_i) \in E$ (bzw. $\{v_{i-1}, v_i\} \in E$), also eine Kante ist ($i = 1, \ldots, k$). Ein Weg heißt *einfach*, falls in der Folge $(v_1, \ldots, v_k)$ kein Knoten mehrfach auftritt. Ein einfacher Weg ist heißt *Zyklus* (oder Kreis), falls $v_0 = v_k$. (Für ungerichetete Graphen muss ferner $k \geq 3$ gelten). Ein Graph heißt *azyklisch* (oder kreisfrei), falls es in ihm keinen Zyklus gibt. (Gerichtete, azyklische Graphen werden im Englischen manchmal DAG genannt; directed acyclic graph).

**Satz.**
DIE KNOTEN JEDES GERICHTETEN AZYKLISCHEN GRAPHEN LASSEN SICH SO DURCHNUMMERIEREN, DASS FÜR JEDE KANTE $(u, v) \in E$ GILT: NUMMER VON $u <$ NUMMER VON $v$.
(MAN NENNT EINE SOLCHE NUMMERIERUNG (ODER ANORDNUNG) DER KNOTEN EINE *topologische Sortierung*).

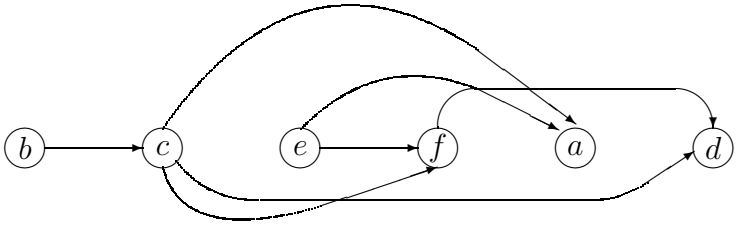
*Beweis:* Induktion über die Anzahl der Knoten. Ein azyklischer Graph mit einem Knoten kann gar keine Kante enthalten und ist unmittelbar topologisch sortiert.

Sei $G$ ein gerichteter, azyklischer Graph mit $n+1$ Knoten. Es muss in $G$ mindestens einen Ausgangsknoten (einen Knoten ohne Nachfolger) geben (sonst könnte man einen Kreis konstruieren). Nennen wir diesen Knoten $v$. Indem wir $v$ aus $G$ entfernen (samt aller zu $v$ führenden Kanten), erhalten wir einen Graphen $G'$ mit $n$ Knoten. Die Knoten von $G'$ können laut Induktionsvoraussetzung von 1 bis $n$ topologisch durchnummeriert werden. Wir übernehmen für $G$ diese Nummerierung und geben $v$ die Nummer $n + 1$. □

*Beispiel:* Der folgende azyklische Graph

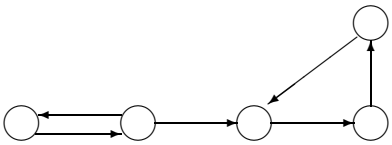



kann zum Beispiel folgendermaßen toplogisch sortiert werden (die Sortierung ist durch die lineare Anordnung der Knoten in der Zeichnung gegeben):




Ein ungerichteter Graph heißt *zusammenhängend*, falls es von jedem Knoten einen Weg zu jedem anderen Knoten in dem Graphen gibt. Handelt es sich um einen gerichteten Graphen, so heißt dieser bei Vorliegen derselben Definition (wobei die Kantenrichtung beachtet werden muss) *stark zusammenhängend*. Ignorieren wir jedoch die Kantenrichtung und fassen den Graphen als ungerichteten auf, und ist dieser dann zusammenhängend, so heißt der ursprüngliche gerichtete Graph *schwach zusammenhängend*.

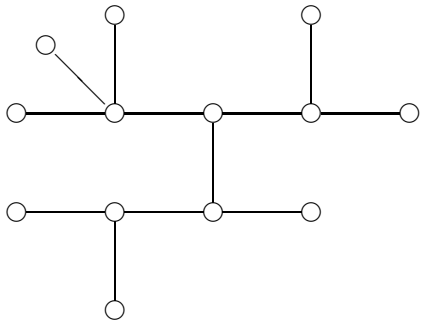
*Beispiel:* Der folgende gerichtete Graph ist schwach aber nicht stark zusammenhängend:



Eine *(starke) Zusammenhangskomponente* ist ein stark zusammenhängender Teilgraph (der nicht weiter vergrößert werden kann). Wenn man die Zusammenhanggskomponenten eines Graphen als Knoten versteht, so muss der zugehörige „Zusammenhangskomponenten-Graph“ azyklisch sein.

Ein ungerichteter, zusammenhängender, azyklischer Graph heißt auch ein *Baum*. (Wird kein Zusammenhang des Graphen verlangt, so spricht man auch von einem *Wald*).

*Beispiel:*

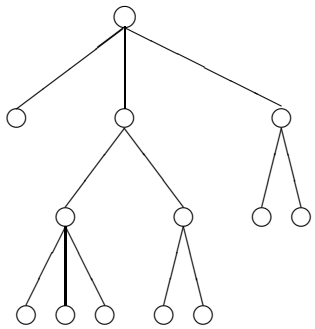


**Satz.**
EIN BAUM MIT $n$ KNOTEN HAT IMMER $n-1$ KANTEN.

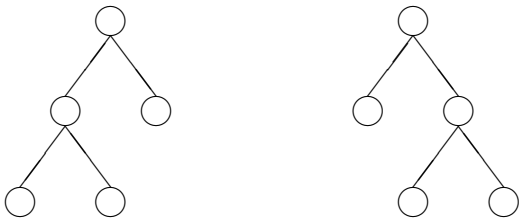
*Beweis:* (Induktion über $n$). Der Fall $n = 1$ ist klar. Sei nun $B$ ein Baum mit $n+1$ Knoten. Wir wählen irgendeinen Knoten $v$ mit genau einem Nachbarn aus (ein solcher Knoten heißt auch *Blatt*). (Wenn es keinen solchen Knoten gäbe, so enthielte $B$ einen Kreis). Indem wir $v$ samt der zu $v$ führenden Kante entfernen, erhalten wir einen Baum mit $n$ Knoten, der nach Induktionsvoraussetzung $n-1$ Kanten hat. Also hat $B$ $n$ Kanten, was zu zeigen war. □

Indem wir irgendeinen Knoten des Baumes besonders hervorheben – und nun *Wurzel* nennen – und alle anderen Knoten entsprechend ihrer Entfernung von der Wurzel als „Söhne“ und „Enkel“, etc. auffassen, erhalten wir einen (für die Informatik sehr typischen) *Wurzelbaum*. (Dies kann durch eine entsprechende Zeichnung – die Wurzel nach oben – hervorgehoben werden).

Das folgende Beispiel ist ein Wurzelbaum (der aus dem obigen Baum durch Identifizierung eines Knotens als Wurzel und entsprechender hierarchisch aufgebauter Zeichnung entsteht):

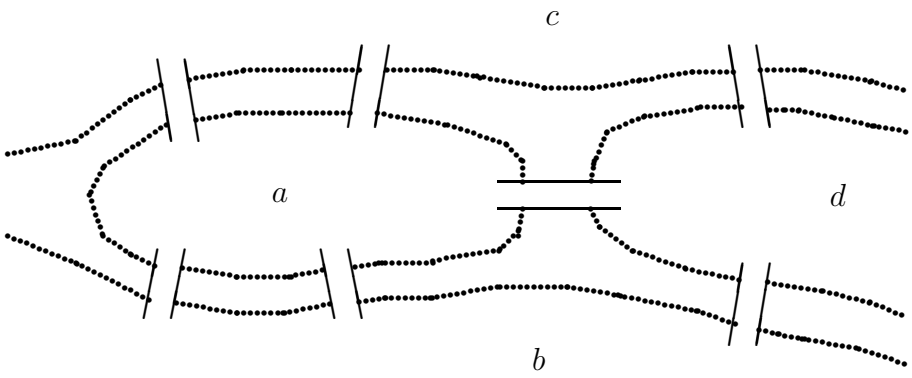


Eine weitere, in der Informatik häufig vorkommende Variante sind *geordnete* Wurzelbäume: hier kommt es auf die Reihenfolge der Söhne (samt der darunterliegenden Teilbäume) an. In diesem Fall sind die folgenden beiden (geordneten Wurzel-) Bäume *nicht* isomorph:



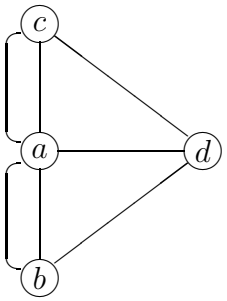
### 2.4.2 Euler- und Hamiltonkreise

Die Stadt Königsberg liegt am Fluss Pregel. Im 18. Jahrhundert führten über die Flussläufe 7 Brücken, die die Gebiete $a, b, c, d$ (vgl. folgendes Bild) miteinander verbanden.




Es war eine beliebte Frage unter den Königsbergern, ob es möglich sei, einen Spaziergang zu machen, der über alle 7 Brücken führt und keine zweimal besucht und zum Ausgangspunkt zurückführt.

Leonard Euler (1707–1783) löste das Problem 1736 (und begründete damit die Graphentheorie). Er war zu der Zeit Mathematik-Professor an der Akademie von Petersburg. Euler reduzierte das Problem auf das Wesentliche; es kommt ja nur auf die Gebiete $a, b, c, d$ und deren möglichen Verbindungen an. Dies ergibt den folgenden Graphen, der ausnahmsweise Mehrfachkanten hat:




Gesucht ist hier also ein Kreis in einem Graphen, der jede *Kante* genau einmal besucht (Knoten dürfen mehr als einmal besucht werden); dies nennt man heutzutage einen *Euler-*

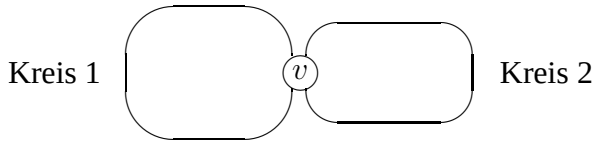
*Kreis*.

Euler bewies ein allgemeines Theorem, aus welchem sich sofort ergibt, dass obiger Graph keinen Euler-Kreis besitzt, dass das Königsberger Brückenproblem also nicht lösbar ist. Dies liegt daran, dass (zum Beispiel) der Knoten $d$ drei Nachbarn besitzt.

**Satz (Euler, 1736).**
EIN ZUSAMMENHÄNGENDER GRAPH (MEHRFACHKANTEN ZUGELASSEN) BESITZT EINEN EULER-KREIS GENAU DANN, WENN DER GRAD JEDES KNOTENS IM GRAPHEN GERADE IST.

*Beweis:* Wenn ein Graph einen Euler-Kreis besitzt, so muss der Grad jedes Knotens gerade sein, denn jeder „Besuch“ eines Knotens erfordert 2 Kanten; eine zum Hineingehen und eine zum Hinausgehen.

Sei nun umgekehrt $G$ ein zusammenhängender Graph mit nur geraden Knotengraden. Wir beginnen in irgendeinem Knoten und durchlaufen einen beliebigen Weg (indem wir einmal benutzte Kanten entfernen). Dies kann nur endlich viele Schritte gutgehen; irgendwann endet der Weg in einer „Sackgasse“. Der Endpunkt kann nur der Ausgangsknoten sein, da dieser der einzige ist, der mit ungeradem Grad „hinterlassen“ wurde. Wir erhalten so also einen Kreis; allerdings muss dieser nicht notwendigerweise alle Kanten des Graphen mit einbeziehen. Wenn der Kreis noch kein Euler-Kreis ist, so muss auf dem Kreis ein Knoten existieren, der noch 2 (oder mehr) „unverbrauchte“ Kanten hat (da $G$ zusammenhängend ist). Wir setzen an diesem Knoten $v$ nochmals auf (nach demselben Verfahren) und erhalten einen weiteren Kreis:

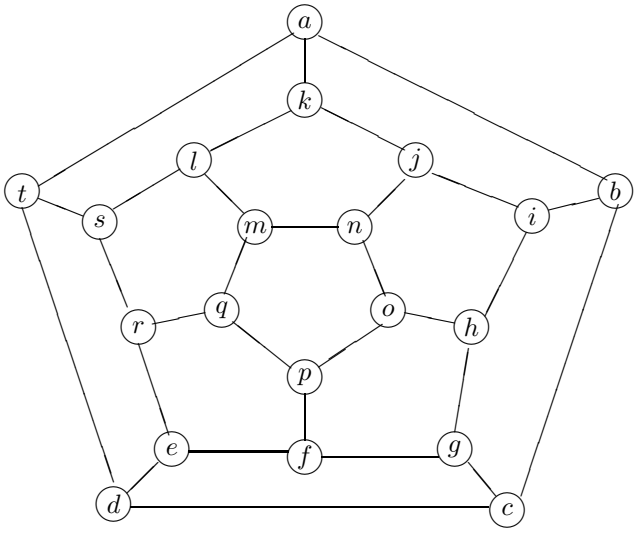



Indem wir diese beiden Kreise in Form einer „Acht“ durchlaufen, haben wir einen größeren Kreis erzeugt. Dieses Verfahren kann man solange fortsetzen, bis alle Kanten abgearbeitet sind und ein Euler-Kreis entstanden ist. □

Der Satz besagt also, dass es algorithmisch sehr einfach ist, festzustellen, ob ein Graph einen Euler-Kreis enthält. Tatsächlich liefert der obige Beweis auch ein effizientes Verfahren, um einen solchen zu konstruieren.

Mit einer ganz ähnlichen Aufgabe hat sich der irische Mathematiker Sir William Rowan Hamilton (1805–1865) befasst. Er erfand 1859 das Knobelspiel „Reise um die Welt“, welches auf einem Dodekaeder gespielt wird (einem „Würfel“ mit 12 Flächen; jede Fläche ist ein regelmäßiges Fünfeck). Die 20 Ecken des Dodekaeders waren mit bekannten Städtenamen beschriftet. Die Aufgabe bestand darin, entlang der Kanten des Dodekaeders zu „reisen“ und jede Stadt genau einmal zu besuchen, bis man zum Ausgangspunkt zurückkehrt.

Wenn wir die Ecken und Kanten des Dodekaeders „flach“ in der Ebene einzeichnen, erhalten wir folgenden Graphen:




Eine mögliche Rundreise erhält man, wenn man die Knoten in der Reihenfolge $a, b, c, \ldots, s, t$, und dann wieder $a$, besucht. Einen derartigen Kreis, auf dem jeder *Knoten* genau einmal vorkommt, nennt man einen *Hamilton-Kreis*.

Man beachte, ein Graph, der einen Euler-Kreis besitzt, muss nicht unbedingt einen Hamilton-Kreis besitzen und ebensowenig umgekehrt. Man kann Beispielgraphen für alle 4 möglichen Situationen (mit/ohne Eulerkreis; mit/ohne Hamilton-Kreis) konstruieren.

Das Problem, von einem Graphen festzustellen, ob er einen Hamilton-Kreis besitzt, ist ungleich schwieriger algorithmisch zu lösen als im Falle des Euler-Kreises. Es sind nur Algorithmen mit exponentieller Laufzeit bekannt.

Es gibt verschiedene hinreichende (aber nicht unbedingt notwendige) Kriterien für einen Graphen, einen Hamilton-Kreis zu besitzen. Insbesondere muss ein Graph dann einen Hamilton-Kreis besitzen, wenn er in gewissem Sinne „viele“ Kanten hat:

**Satz (Dirac, 1952).**
WENN $G$ EIN ZUSAMMENHÄNGENDER GRAPH MIT $n \geq 3$ KNOTEN IST, IN DEM JEDER KNOTEN MINDESTENS $n/2$ VIELE NACHBARN HAT, DANN HAT $G$ EINEN HAMILTON-KREIS.

Dieser Satz ergibt sich unmittelbar aus dem folgenden Satz.

**Satz (Ore, 1960).**
WENN $G$ EIN ZUSAMMENHÄNGENDER GRAPH MIT $n \geq 3$ KNOTEN IST, IN DEM FÜR ZWEI NICHT-BENACHBARTE KNOTEN $u, v$ IMMER GILT $d(u) + d(v) \geq n$, DANN HAT $G$ EINEN HAMILTON-KREIS.

*Beweis:* Sei $(v_1, \ldots, v_n)$ eine Permutation der Knoten in $G$. Wir setzen $v_0 = v_n$. Ein Paar $(v_{k-1}, v_k)$ heiße eine *Lücke*, wenn $v_{k-1}$ und $v_k$ in $G$ nicht benachbart sind. Unter den möglichen $n!$ möglichen Permutationen wählen wir für das Folgende eine Permutation mit minimaler Lückenzahl. Wir zeigen nun, dass im Falle einer Lücke $(v_{k-1}, v_k)$ die Summe der Grade von $v_{k-1}$ und $v_k$ höchstens $n-1$ ist. Gemäß der Voraussetzung des Satzes gibt es dann also gar keine Lücken, und somit ist $(v_0, v_1, \ldots, v_n)$ ein Hamilton-Kreis.

Wenn ein Knoten $v_j$ Nachbar von $v_{k-1}$ ist, dann kann $v_{j+1}$ kein Nachbar von $v_k$ sein, denn sonst wäre im Falle von $j > k$

$$v_1, v_2, \ldots, v_{k-1}, v_j, v_{j-1}, \ldots, v_k, v_{j+1}, v_{j+2}, \ldots, v_n$$

und im Falle von $j < k$

$$v_1, v_2, \ldots, v_j, v_{k-1}, v_{k-2}, \ldots, v_{j+1}, v_k, v_{k+1}, \ldots, v_n$$

eine Permutation mit einer Lücke weniger, was der Minimalität unserer gewählten Permutation widerspricht. Zu jedem Nachbarn von $v_{k-1}$ kann also eineindeutig ein Nicht-Nachbar von $v_k$ angegeben werden. Sei $l = d(v_{k-1})$, dann sind für $v_k$ gerade $l$ potentielle Nachbarn ausgeschlossen. Die Summe der Grade von $v_{k-1}$ und $v_k$ ist daher höchstens $l + (n - 1 - l) = n - 1$. □

Eng verwandt mit dem Problem, einen Hamilton-Kreis in einem Graphen zu bestimmen, ist das *Traveling Salesman Problem* (oder Problem des Handlungsreisenden). Es unterscheidet sich vom Hamilton-Kreis-Problem nur dadurch, dass den Kanten des Graphen Zahlen zugeordnet sind (sozusagen: Entfernungen zwischen den zu besuchenden Städten). Die Aufgabe besteht darin, eine Rundreise (also einen Hamilton-Kreis) zu finden, die die Summe der zurückgelegten Entfernungen minimiert. Für dieses Problem sind auch nur exponentielle Algorithmen bekannt.

### 2.4.3 Adjazenzmatrizen

Wie man beim Traveling Salesman Problem gesehen hat, sind viele Fragen über die Existenz von Wegen und Kreisen in Graphen sehr wichtig und haben praktische Anwendungen. Wir wollen ein paar grundsätzliche Fragen über die Existenz von Wegen in gerichteten Graphen mit Hilfe sog. *Adjazenzmatrizen* behandeln.

Die einem gerichteten Graphen $G = (V, E)$ mit $V = \{1, 2, \ldots, n\}$ zugeordnete Adjazenzmatrix $M_G$ ist eine Boolesche $n \times n$ Matrix, das heißt, die Einträge sind 0 oder 1; und zwar wird in Position $(i, j)$ genau dann eine 1 eingetragen, wenn die Kante $(i, j)$ vorhanden ist.

Betrachten wir die Boolesche Matrizenmultiplikation von $M_G = (m_{ij})$ mit sich selbst. In der Matrix $M_G \cdot M_G$ ist an der Stelle $(i, j)$ nach Definition der Matrizenmultiplikation genau dann eine 1, wenn $\bigvee_{k=1}^{n}(m_{ik} \wedge m_{kj}) = 1$. Das ist in Worten genau dann der Fall, wenn es einen Knoten $k$ gibt, so dass die Kante $(i, k)$ und die Kante $(k, j)$ im Graphen

existieren. Nochmals in anderen Worten heißt dies, dass es im Ausgangsgraphen einen Weg der Länge 2 von $i$ nach $j$ gibt.

Verallgemeinert heißt dies, dass $M_G^{(m)}$ (das $m$-fache Matrizenprodukt von $M_G$ mit sich selbst) an der Stelle $(i, j)$ eine 1 hat, genau dann wenn es in $G$ einen Pfad der Länge $m$ von $i$ nach $j$ gibt.

Als eine mögliche Anwendung dieser Beobachtung erhalten wir sofort den folgenden Satz:

**Satz.**
EIN GERICHTETER GRAPH $G$ HAT GENAU DANN EINEN KREIS, WENN FÜR ALLE $k \in \{1, \ldots, n\}$ GILT $M_G^{(k)} \neq \mathbf{0}$. (HIERBEI IST $\mathbf{0}$ DIE ENTSPRECHEND DIMENSIONIERTE NULLMATRIX).

*Beweis:* Wenn $G$ zyklisch ist, so hat $G$ Wege beliebiger Länge. Insbesondere ist für $k = 1, \ldots, n$ nach der obigen Beobachtung: $M_G^{(k)} \neq \mathbf{0}$.

Es gelte nun für jedes $k \leq n$, dass $M_G^{(k)} \neq \mathbf{0}$. Insbesondere gilt dies für $k = n$. Sei an der Stelle $(i, j)$ von $M_G^{(n)}$ der Wert =1. Dann gibt es zwischen Knoten $i$ und Knoten $j$ einen Weg der Länge $n$, zum Beispiel $(i = v_0, v_1, \ldots, v_n = j)$. Da die Knotenmenge $V$ aber nur die Mächtigkeit $n$ hat, müssen nach dem Schubfachprinzip zwei der Knoten in der Folge $(v_0, v_1, \ldots, v_n)$ übereinstimmen. Sei etwa $v_k = v_{k+r}$ mit $r > 0$. Dann ist $(v_k, v_{k+1}, \ldots, v_{k+r})$ ein Kreis. □

Auskunft über alle möglichen Wege-Verbindungen in einem Graphen $G$ gibt die Matrix

$$M_G \cup M_G^{(2)} \cup \ldots \cup M_G^{(n)}$$

In der Sprechweise der Relationen ist dies nichts anderes als die transitive Hülle von $G$ – und zwar berechnet nach der Methode „von unten nach oben“ ; vgl. den Abschnitt über Hüllenbildungen.

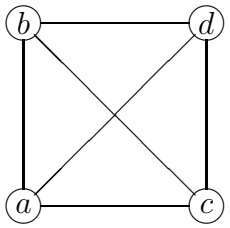
Eine effiziente Methode (Komplexität $O(n^3)$) zur Berechnung der Wege-Matrix (=transitive Hülle) stellt der *Algorithmus von Warshall* dar:

```
FOR k:=1 TO n DO
   FOR i:=1 TO n DO
      IF M[i,k] THEN
         FOR j:=1 TO n DO
            IF M[k,j] THEN M[i,j]:=TRUE END;
         END;
      END;
   END;
END;
```

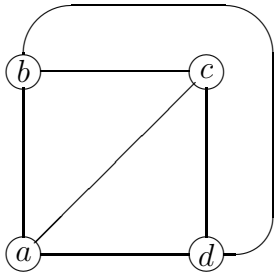
### 2.4.4 Planare Graphen

Ein Graph heißt *planar*, wenn er sich in der Ebene so zeichnen lässt, dass sich keine zwei Kanten überkreuzen. (Kanten dürfen hierbei, wenn nötig, als krumme Linien gezeichnet werden).

*Beispiel:* Der folgende Graph ist planar




denn er kann wie folgt gezeichnet werden:




Weiteres Beispiel: Der Dodekaeder-Graph auf Seite 45 ist planar, wie die Zeichnung zeigt.

In vielen Anwendungen kommen in natürlicher Weise planare Graphen vor oder sind planar gezeichnete Graphen besonders wünschenswert. (Beispiel: Beim Design von elektrischen Leiterplatten oder beim VLSI sollten sich Leitungen nicht überkreuzen).

Wird ein planarer Graph in der Ebene kreuzungsfrei gezeichnet (was auf verschiedene Arten möglich sein kann), so wird die Zeichenebene hierbei in verschiedene *Regionen* (oder Flächen) zerlegt, wobei die außenliegende Region unendlich groß ist und alle innenliegenden Regionen endlich groß sind. Der folgende Satz von Euler besagt, dass es für die Anzahl der Regionen keine Rolle spielt, wie der Graph gezeichnet ist, und ferner stellt er einen Zusammenhang zwischen Regionenzahl, Knotenzahl und Kantenzahl her.

**Satz (Euler, 1752).**
IN JEDEM ZUSAMMENHÄNGENDEN, PLANAREN GRAPHEN GILT:
KNOTENZAHL − KANTENZAHL + REGIONENZAHL = 2

Im Beispielgraph oben gilt: Knotenzahl=4, Kantenzahl=6, Regionenzahl=4.

Im Dodekaedergraph (Seite 45) gilt: Knotenzahl=20, Kantenzahl=30, Regionenzahl=12.

*Beweis:* Im folgenden sei immer $n$ die Knotenzahl, $m$ die Kantenzahl und $r$ die Regionenzahl.

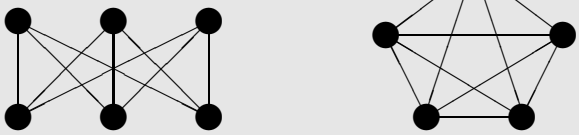
Wir stellen uns vor, das kreuzungsfreie Zeichnen eines zusammenhängenden, planaren Graphen geschehe schrittweise. Als mögliche Schritte kommen in Frage:

1. (Als erster Schritt) Das Plazieren eines Knotens;
2. Die Unterteilung einer Kante durch einen neuen Knoten;
3. Die Verbindung zweier schon vorhandener Knoten durch eine Kante, die hierbei keine andere Kante schneidet;
4. Das Ansetzen einer neuen Kante mit einem neuen Endknoten an einen schon vorhandenen Knoten.

Nach dem ersten Schritt haben wir $r = n = 1$, $m = 0$, also $n - m + r = 2$. Bei 2. erhöhen sich $n$ und $m$ um 1; genauso ist es bei 4.; der Wert von $n - m + r$ bleibt also unverändert (nämlich 2). Bei 3. erhöhen sich $m$ und $r$ um 1; also bleibt auch hier der Wert von $n - m + r$ konstant. □

Wir zitieren ohne Beweis ein berühmtes Ergebnis der Graphentheorie, das die planaren Graphen exakt charakterisiert:

**Satz (Kuratowski, 1930).**
EIN GRAPH IST GENAU DANN PLANAR, WENN ER NACH ENTFERNEN „ÜBERFLÜSSIGER KNOTEN“ KEINEN DER FOLGENDEN BEIDEN TEILGRAPHEN ENTHÄLT:



„ÜBERFLÜSSIGE KNOTEN ENTFERNEN“ BEDEUTET, DASS MAN JEDEN KNOTEN, DER GENAU ZWEI NACHBARN HAT, ENTFERNT (ABER DIE KANTENVERBINDUNG AUFRECHT ERHÄLT).

Mit Hilfe dieses Satzes lassen sich effiziente Algorithmen zum Testen der Planarität eines Graphen konstruieren.

### 2.4.5 Färbbarkeit

Eine *Färbung* eines Graphen $G = (V, E)$ mit $k$ Farben ist eine Abbildung $f : V \longrightarrow \{1, \ldots, k\}$, so dass für alle Kanten $\{u, v\} \in E$ gilt $f(u) \neq f(v)$.

Die *chromatische Zahl* eines Graphen $G$, die wir mit $\chi(G)$ bezeichnen, ist definiert als das kleinste $k$, so dass $G$ mit $k$ Farben färbbar ist.

Eine triviale obere Schranke für die chromatische Zahl eines Graphen $G$ ist die Anzahl der Knoten in $G$. Falls $K_l$, der vollständige Graph mit $n$ Knoten (das heißt, alle $\binom{l}{2}$ Kanten sind vorhanden), ein Teilgraph von $G$ ist, dann gilt: $\chi(G) \geq l$. Umgekehrt muss aber ein Graph mit $\chi(G) = l$ keinen vollständigen Graphen mit $l$ Knoten enthalten.

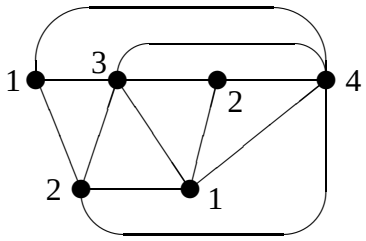
Eine Teilmenge $C$ der Knoten eines ungerichteten Graphen $G$ heißt *Clique*, falls der von $C$ induzierte Teilgraph in $G$ vollständig ist, d.h., für alle Knotenpaare $u, v$ in $C$ $\{u, v\}$ eine Kante in $G$ ist. Die *Cliquenzahl* eines Graphen $G$, $\omega(G)$, ist die Mächtigkeit einer Clique von $G$ mit den meisten Knoten. Es gilt:

$$\chi(G) \geq \omega(G).$$

Die chromatische Zahl oder die Cliquenzahl eines Graphen zu bestimmen sind algorithmisch schwierige Aufgaben. Hierfür sind nur exponentielle Algorithmen bekannt.

Auf das Jahr 1852 geht das sog. *Vier-Farben-Problem* zurück. Es stellt die Frage, ob die Länder einer jeden Landkarte so mit 4 Farben eingefärbt werden kann, dass keine zwei benachbarten Länder dieselbe Farbe erhalten. Wenn wir jedes Land mit einem Knoten identifizieren und Nachbarschaft durch eine Kante ausdrücken, so entsteht (sozusagen automatisch) ein *planarer* Graph. Die Frage lautet also nun, ob für jeden planaren Graph $G$ gilt $\chi(G) \leq 4$; also ob jeder planare Graph 4-färbbar ist.

*Beispiel:* Der folgende planare Graph ist 4-färbbar, aber nicht 3-färbbar (eine mögliche Färbung mit 4 Farben ist eingetragen).




Nach über hundert Jahren erfolglosen Bemühens um diese Frage wurde das Vier-Farben-Problem schließlich 1976 von Appel und Haken gelöst. Und zwar zeigten diese, dass man „nur“ eine bestimmte endliche Anzahl von Graphen vollständig analysieren muss. Diese endlich vielen Fälle belaufen sich auf etwa 2000 Stück. Diese wurden dann per Computer in über 1200 Stunden CPU-Zeit analysiert und das Ergebnis war jedesmal positiv. Appel und Hakens Beweis besteht also zum größten Teil aus vielen hundert Seiten Computerpapier. Man kann sich vorstellen, dass das Akzeptieren der Tatsache, dass kein Mensch diesen Beweis mehr „in endlicher Zeit“ von Hand nachprüfen kann (und dies trotzdem ein Beweis ist), vielen Mathematikern erhebliche Kopfschmerzen bereitet.

Aus verständlichen Gründen geben wir den Vier-Farben-Satz hier *ohne* Beweis an:

**Satz (K. Appel, W. Haken, 1976).**
JEDER PLANARE GRAPH IST 4-FÄRBBAR.

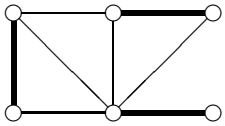
*Bemerkung:* Ein entsprechender 5-Farben-Satz ist ganz einfach beweisbar (vgl. irgendein Buch über Graphentheorie).

Eine interessante Klasse von Graphen bilden die 2-färbbaren Graphen, auch *bipartite* Graphen genannt. Bei diesen Graphen kann man die Knotenmenge $V$ disjunkt in 2 Teilmengen $V_1$ und $V_2$ zerlegen; hierbei sind $V_1$ gerade diejenigen Knoten, die mit Farbe 1 gefärbt sind und $V_2$ diejenigen, die mit Farbe 2 gefärbt sind. Jede vorkommende Kante verbindet einen Knoten in $V_1$ mit einem Knoten in $V_2$.

### 2.4.6 Matchings

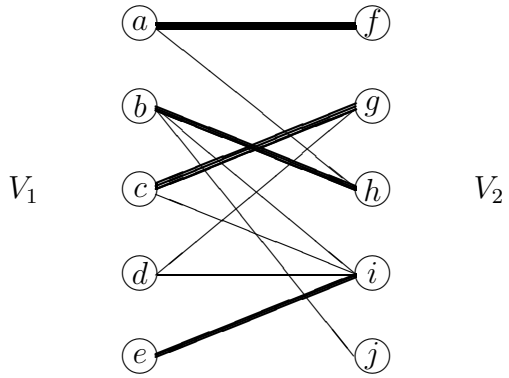
Ein *Matching* in einem Graphen ist eine Teilmenge $M \subseteq E$ der Kanten, so dass keine zwei Kanten einen Endknoten gemeinsam haben. Ein Matching $M$ heißt *perfekt*, falls durch die Kanten in $M$ *alle* Knoten des Graphen erfasst werden.

*Beispiel:* Die folgenden fett gezeichneten Kanten bilden ein perfektes Matching:



Das Problem, ein perfektes (oder möglichst großes) Matching zu bestimmen, ist besonders bei bipartiten Graphen interessant.

Betrachten wir folgenden bipartiten Graphen mit den disjunkten Knotenmengen $V_1$ und $V_2$. Ein mögliches Matching, bestehend aus 4 Kanten, ist mit fetten Linien eingezeichnet.




Theoretisch denkbar wäre in diesem Fall ein Matching mit 5 Kanten, da $V_1$ und $V_2$ je 5 Knoten haben. Das wäre dann ein perfektes Matching. Dies ist jedoch bei dem Beispiel oben unmöglich, da die 3 linken Knoten $c, d, e$ insgesamt nur die 2 Nachbarn $g, i$ haben. Einer der drei linken Knoten wird also nicht „gematcht“ werden können, egal wie man es anstellt.

Mit $N(A)$ bezeichnen wir die Menge der Nachbarn einer Knotenmenge $A$. Um ein perfektes Matching zu erhalten, ist es eine notwendige Bedingung, dass $|N(A)| \geq |A|$ für

jede Teilmenge $A$ der linken Knotenmenge $V_1$ gilt.

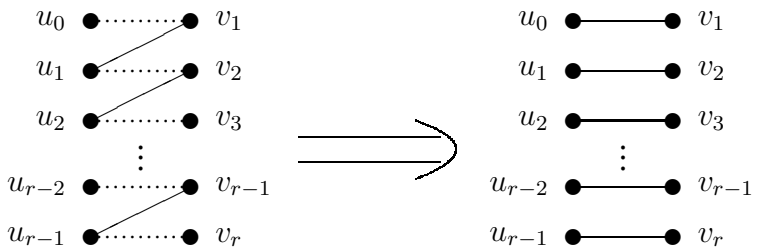
Erstaunlicherweise ist diese Bedingung auch *hinreichend*. Dies ist die Aussage des folgenden Satzes (bekannt unter dem Namen *Heiratssatz* wegen folgender Interpretation: linke Knotenmenge=Menge von Damen; rechte Knotenmenge=Menge von Herren; Kante vorhanden=Heirat nicht ausgeschlossen).

**Satz (Hall, 1935).**
SEI DER BIPARTITE GRAPH $G = (V_1 + V_2, E)$ GEGEBEN. DANN GIBT ES EIN MATCHING $M$ MIT $|M| = |V_1|$ GENAU DANN, WENN $|N(A)| \geq |A|$ FÜR ALLE TEILMENGEN $A$ VON $V_1$ GILT.

*Beweis:* Dass $|N(A)| \geq |A|$ notwendigerweise bei Vorhandensein eines maximalen Matchings gilt, ist klar.

Sei nun umgekehrt die Bedingung $|N(A)| \geq |A|$ erfüllt. Zu jedem gegebenen Matching $M \subset E$ mit $|M| < |V_1|$ zeigen wir, dass $|M|$ nicht maximal ist. Sei $u_0 \in V_1$ ein Knoten, der durch $M$ nicht gematcht wird. Da $|N(\{u_0\})| \geq |\{u_0\}| = 1$, existiert mindestens ein Nachbar $v_1$ in $V_2$. Wir wählen möglichst einen nicht an $M$ beteiligten Knoten $v_1$ und fügen dann einfach die Kante $\{u_0, v_1\}$ zu $M$ hinzu und erhalten ein größeres Matching.

Nehmen wir also an, dass dies nicht möglich ist und $\{u_1, v_1\} \in M$ ist für ein $u_1 \neq u_0$. Da $|N(\{u_0, u_1\})| \geq |\{u_0, u_1\}| = 2$, gibt es einen Knoten $v_2 \neq v_1$, welcher zu $u_0$ oder $u_1$ benachbart ist. Falls $v_2$ nicht in $M$ beteiligt ist, können wir ähnlich wie oben $M$ (umarrangieren und) erweitern. Andernfalls existiert eine Kante $\{u_2, v_2\} \in M$ mit $u_2 \notin \{u_0, u_1\}$. Wir fahren auf diese Weise fort und erreichen schließlich einen nicht in $M$ beteiligten Knoten $v_r \in V_2$, $r \leq |M| < |V_1|$. Der Knoten $v_r$ hat eine Kante zu $u_{r-1}$. Nun kann das bisherige Matching wie folgt umarrangiert und um eine Kante vergrößert werden. (Hierbei deuten gestrichelte Linien bestehende, aber nicht am Matching beteiligte Kanten an; durchgezogene Linien sind am Matching beteiligt).




□

Für das Problem zu entscheiden, ob es in einem bipartiten Graphen $G = (V_1 + V_2, E)$ ein Matching $M$ mit $|M| = |V_1|$ gibt, sind effiziente Algorithmen bekannt. Diese verwenden Ideen aus dem Beweis des Heiratssatzes.

# Teil 3

# Grammatiken und Automaten

## 3.1 Allgemeines

Sei $\Sigma$ ein *Alphabet*, also eine endliche Menge, deren Elemente wir als *Buchstaben* oder *Symbole* bezeichnen. Eine (formale) *Sprache* (über $\Sigma$) ist jede beliebige Teilmenge von $\Sigma^*$.

Sei z.B. $\Sigma = \{(, ), +, -, *, /, a\}$, so könnten wir die Sprache der korrekt geklammerten arithmetischen Ausdrücke $EXPR \subseteq \Sigma^*$ definieren, wobei $a$ als Platzhalter für beliebige Konstanten oder Variablen dienen soll:

$$\begin{aligned} (a-a)*a+a/(a+a)-a &\in EXPR \\ (((a))) &\in EXPR \\ ((a+)-a( &\notin EXPR \end{aligned}$$

Um mit solchen Sprachen, die im Allgemeinen *unendliche* Objekte sind, algorithmisch umgehen zu können, benötigen wir jedoch *endliche* Beschreibungsmöglichkeiten für Sprachen. Dazu dienen sowohl die *Grammatiken* als auch die *Automaten*.

*Beispiel* für eine Grammatik:

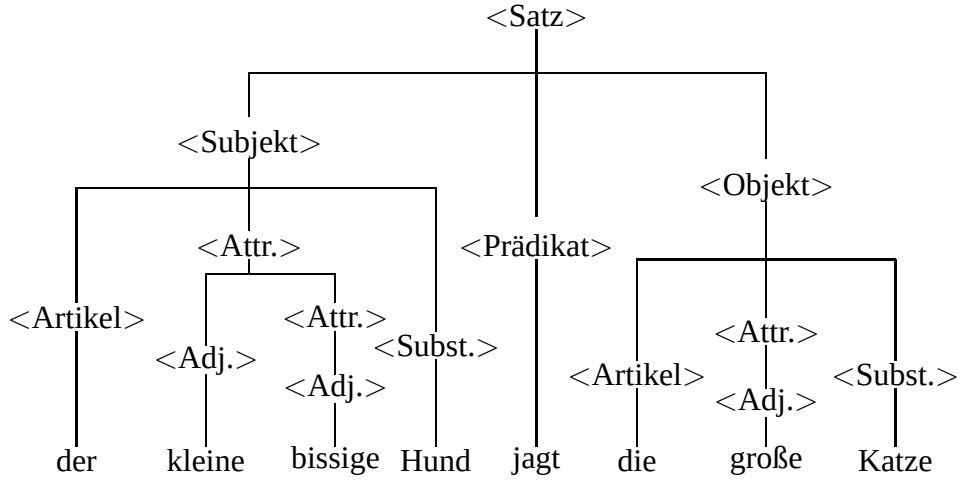
| | | |
|---|---|---|
| <Satz> | → | <Subjekt> <Prädikat> <Objekt> |
| <Subjekt> | → | <Artikel> <Attribut> <Substantiv> |
| <Artikel> | → | $\varepsilon$ |
| <Artikel> | → | der |
| <Artikel> | → | die |
| <Artikel> | → | das |
| <Attribut> | → | $\varepsilon$ |
| <Attribut> | → | <Adjektiv> |
| <Attribut> | → | <Adjektiv> <Attribut> |
| <Adjektiv> | → | kleine |
| <Adjektiv> | → | bissige |
| <Adjektiv> | → | große |
| <Substantiv> | → | Hund |
| <Substantiv> | → | Katze |
| <Prädikat> | → | jagt |
| <Objekt> | → | <Artikel> <Attribut> <Substantiv> |

Hierbei sind die sog. *Variablen*, die Platzhalter für syntaktische Einheiten sind, durch spitze Klammern kenntlich gemacht.

Durch die obigen Grammatik-Regeln kann z.B. der Satz

der kleine bissige Hund jagt die große Katze

abgeleitet werden. Besonders anschaulich kann dies durch einen *Syntaxbaum* dargestellt werden. Hierbei ist der Vaterknoten jeweils mit der linken Seite einer Regel beschriftet und seine Söhne sind die Objekte, die auf der rechten Seite der Regel stehen.




Man beachte, dass durch diese endliche Grammatik bereits eine unendliche Sprache darstellbar ist, denn es sind z.B. alle Sätze der Form

der Hund jagt die kleine kleine kleine ... Katze

erzeugbar.

### 3.1.1 Grammatiken

Wie sieht nun im Hinblick auf eine allgemeine Definition eine Grammatik aus? Zunächst muss angegeben werden, welches Variablen und welches „eigentliche“ Symbole sind (sog. *Terminalsymbole*). Sodann müssen die Regeln angegeben werden, die allgemein die Form

$$\textit{linke Seite} \rightarrow \textit{rechte Seite}$$

haben. Das obige Beispiel ist insofern ein Spezialfall, da die linken Seiten immer nur aus einer einzelnen Variablen bestehen. (Es handelt sich um eine sog. *kontextfreie* Grammatik). Allgemeiner könnte man zulassen, dass die linken Seiten aus Wörtern bestehen, die sowohl (evtl. mehrere) Variablen, als auch Terminalsymbole enthalten. Die *Anwendung* einer Regel bedeutet dann, dass in dem insoweit erzeugten Wort ein Teilwort, das einer linken Regelseite entspricht, durch die rechte Seite ersetzt wird. Solche Ableitungsschritte werden solange durchgeführt, bis das entstandene Wort nur noch aus Terminalsymbolen besteht. Jedes solcherart erzeugbare Wort gehört dann zu der von der Grammatik erzeugten (oder definierten) Sprache. Eine solche Ableitung, die mit einem Terminalwort endet, beginnt mit einer ausgezeichneten Variablen, der *Startvariablen* (im obigen Beispiel ist es <Satz>).

**Definition.** Eine *Grammatik* ist ein 4-Tupel $G = (V, \Sigma, P, S)$, das folgende Bedingungen erfüllt. $V$ ist eine endliche Menge, die Menge der *Variablen*. $\Sigma$ ist eine endliche Menge, das *Terminalalphabet*. Es muss gelten: $V \cap \Sigma = \emptyset$. $P$ ist die endliche Menge der *Regeln* oder *Produktionen*. Formal ist $P$ eine endliche Teilmenge von $(V \cup \Sigma)^+ \times (V \cup \Sigma)^*$. $S \in V$ ist die *Startvariable*.

Seien $u, v \in (V \cup \Sigma)^*$. Wir definieren die Relation $u \Rightarrow_G v$ (in Worten: $u$ geht unter $G$ unmittelbar über in $v$), falls $u$ und $v$ die Form haben

$$\begin{array}{lcl} u & = & xyz \\ v & = & xy'z \quad \text{mit } x, z \in (V \cup \Sigma)^* \end{array}$$

und $y \rightarrow y'$ eine Regel in $P$ ist. Falls klar ist, welche Grammatik $G$ gemeint ist, so schreiben wir einfach $u \Rightarrow v$ anstatt $u \Rightarrow_G v$.

Die von $G$ dargestellte (erzeugte, definierte) *Sprache* ist

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$$

Hierbei ist $\Rightarrow_G^*$ die reflexive und transitive Hülle von $\Rightarrow_G$.

Eine Folge von Wörtern $(w_0, w_1, \ldots, w_n)$ mit $w_0 = S$, $w_n \in \Sigma^*$ und $w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$ heißt *Ableitung* von $w_n$. Ein Wort $w \in (V \cup \Sigma)^*$, das also noch Variablen enthält – wie es typischerweise im Verlauf einer Ableitung auftritt – heißt auch *Satzform*.

*Beispiel:* $G = (\{E, T, F\},\ \{(, ), a, +, *\},\ P,\ E)$ wobei

$$\begin{aligned} P \ = \ \{\ & E \to T, \\ & E \to E + T, \\ & T \to F, \\ & T \to T * F, \\ & F \to a, \\ & F \to (E)\ \} \end{aligned}$$

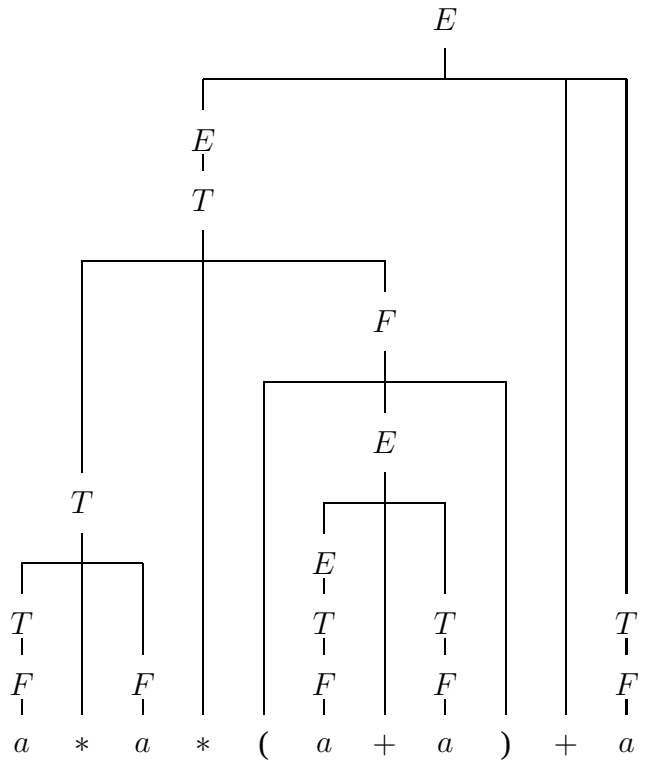
Mit dieser Grammatik lassen sich die korrekt geklammerten arithmetischen Ausdrücke darstellen. Es gilt z.B.

$$a * a * (a + a) + a \ \in \ L(G)$$

denn:

$$\begin{aligned} & E \Rightarrow E + T \Rightarrow T + T \Rightarrow T * F + T \Rightarrow T * F * F + T \Rightarrow \\ & F * F * F + T \Rightarrow a * F * F + T \Rightarrow a * a * F + T \Rightarrow \\ & a * a * (E) + T \Rightarrow a * a * (E + T) + T \Rightarrow a * a * (T + T) + T \Rightarrow \\ & a * a * (F + T) + T \Rightarrow a * a * (a + T) + T \Rightarrow a * a * (a + F) + T \Rightarrow \\ & a * a * (a + a) + T \Rightarrow a * a * (a + a) + F \Rightarrow a * a * (a + a) + a \end{aligned}$$

Hierbei wurde in jedem Ableitungsschritt immer die am weitesten links stehende Variable ersetzt (*Linksableitung*). Ein entsprechender Syntaxbaum sieht folgendermaßen aus:



Ein weiteres *Beispiel*:

$$\begin{aligned} G &= (V, \Sigma, P, S)\text{, wobei:} \\ V &= \{S, B, C\} \\ \Sigma &= \{a, b, c\} \\ P &= \{S \to aSBC, S \to aBC, CB \to BC, \\ &\qquad aB \to ab, bB \to bb, bC \to bc, cC \to cc\} \end{aligned}$$

Es gilt zum Beispiel:

$$\begin{aligned} &S \Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow aaaBCBCBC \Rightarrow \\ &aaaBBCCBC \Rightarrow aaaBBCBCC \Rightarrow aaaBBBCCC \Rightarrow \\ &aaabBBCCC \Rightarrow aaabbBCCC \Rightarrow aaabbbCCC \Rightarrow \\ &aaabbbcCC \Rightarrow aaabbbccC \Rightarrow aaabbbccc = a^3b^3c^3 \end{aligned}$$

Wir vermuten, dass allgemein gilt:

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}$$

Tatsächlich lässt sich dies beweisen:

($\supseteq$) Der oben angegebene Fall $n = 3$ lässt sich leicht für beliebige $n \geq 1$ verallgemeinern.

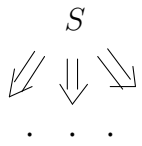
($\subseteq$) Wir beobachten zunächst, dass alle Regeln die „Balance“ erhalten, in dem Sinne, dass in jedem Ableitungsschritt die Anzahl der a's (bzw. A's) gleich der Anzahl der b's (bzw. B's) gleich der Anzahl der c's (bzw. C's) ist. Deshalb muss für jedes Wort $w \in L(G)$ gelten: Anzahl der a's = Anzahl der b's = Anzahl der c's.

Als nächstes inspizieren wir die Regeln im Einzelnen, und zwar im Hinblick darauf, in welcher Reihenfolge diese Terminalsymbole erzeugt werden können. Wir wollen zeigen, dass für jedes $x$ in $L(G)$ gilt: die $a$'s in $x$ kommen vor den $b$'s, und diese vor den $c$'s. Die $a$'s können nur durch die ersten beiden Regeln erzeugt werden und stehen dann, wie gewünscht, ganz links. Betrachten wir nun die $b$'s und $c$'s: Ein nur aus den Terminalzeichen $b$ und $c$ bestehendes Teilwort von $x$ kann nur durch die Regeln $aB \to ab$, $bB \to bb$, $bC \to bc$, $cC \to cc$ erzeugt werden. Diese Regeln sind so aufgebaut, dass die $b$'s sich an die $a$'s anschließen müssen, und dann die $c$'s an die $b$'s.

Beide Beobachtungen zusammengenommen ergeben, dass jedes Wort in $L(G)$ nur die Form $a^n b^n c^n$, $n \geq 1$, haben kann. □

*Bemerkung:* Man beachte, dass das Ableiten kein eindeutiger, *deterministischer* Prozess ist, sondern ein *nichtdeterministischer*. Mit anderen Worten: Die Relation $\Rightarrow_G$ ist i.a. keine Funktion. Für ein gegebenes Wort $x$ aus $(V \cup \Sigma)^*$ kann es mehrere Wörter $x'$ (aber nur endlich viele) geben mit: $x \Rightarrow_G x'$. Zum einen kann es in $x$ mehrere Teilwörter geben, die linke Seite einer Regel sind. Zum anderen kann es für dasselbe Teilwort von $x$ mehrere Regeln mit dieser linken Seite geben.

Graphisch dargestellt kann man sich diese Situation wie einen (endlich verzweigten, aber i.a. unendlich großen) Baum vorstellen mit $S$ an der Wurzel (nicht zu verwechseln mit einem Syntaxbaum):




Den Blättern dieses Baumes sind dann die Wörter der erzeugten Sprache zugeordnet, es kann jedoch auch unendlich lange Pfade geben. Ebenfalls kann es Pfade geben, die in einer Satzform enden, welche nicht mehr weiter zu einem Terminalwort abgeleitet werden kann („Sackgasse“).

*Notation:* Wir werden im Folgenden Grammatiken i.a. nicht so ausführlich angeben, wie es die Definition eigentlich erfordert, sondern uns oft auf die Angabe der Regeln $P$ beschränken. Hierbei gehen wir von folgenden Konventionen aus. Großbuchstaben (oder Wörter in spitzen Klammern) bezeichnen Variablen; Terminalzeichen sind im Allgemeinen an der Kleinschreibung zu erkennen.

### 3.1.2 Chomsky-Hierarchie

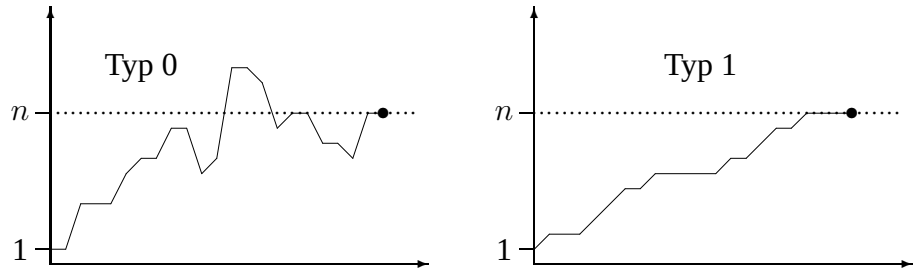
Von Noam Chomsky, einem Pionier der Sprach-Theorie, stammt folgende Einteilung von Grammatiken in *Typen*, nämlich Typ 0 – 3 (manchmal auch Chomsky 0 – 3 genannt).

> **Definition.** Jede Grammatik ist zunächst automatisch vom *Typ 0*. Das heißt, bei Typ 0 sind den Regeln keinerlei Einschränkungen auferlegt. (Man spricht auch von allgemeinen *Phrasenstrukturgrammatiken*).
>
> Eine Grammatik ist vom *Typ 1* oder *kontextsensitiv*, falls für alle Regeln $w_1 \to w_2$ in $P$ gilt: $|w_1| \leq |w_2|$.
>
> Eine Typ 1–Grammatik ist vom *Typ 2* oder *kontextfrei*, falls für alle Regeln $w_1 \to w_2$ in $P$ gilt, dass $w_1$ eine einzelne Variable ist, d.h. $w_1 \in V$.
>
> Eine Typ 2–Grammatik ist vom *Typ 3* oder *regulär*, falls zusätzlich gilt: $w_2 \in \Sigma \cup \Sigma V$, d.h. die rechten Seiten von Regeln sind entweder einzelne Terminalzeichen oder ein Terminalzeichen gefolgt von einer Variablen.
>
> Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ 0 (Typ 1, Typ 2, Typ 3), falls es eine Typ 0 (Typ 1, Typ 2, Typ 3)–Grammatik $G$ gibt mit $L(G) = L$.

*Bemerkung:* Die obige Beispielgrammatik für $a^n b^n c^n$ ist kontextsensitiv (Typ 1), die Grammatik für die arithmetischen Ausdrücke ist kontextfrei (Typ 2).

Die Bezeichnungen „kontextfrei“ und „kontextsensitiv“ haben folgende Begründung: Bei Vorliegen einer kontextfreien Regel $A \to x$ kann die Variable $A$ – unabhängig vom Kontext, in dem $A$ steht – bedingungslos durch $x$ ersetzt werden. Bei einer kontextsensitiven Grammatik dagegen ist es möglich, Regeln der Form $uAv \to uxv$ anzugeben. Das bedeutet, dass $A$ nur dann durch $x$ ersetzt werden kann, wenn die Variable $A$ im „Kontext“ zwischen $u$ und $v$ steht.

Das folgende Bild veranschaulicht für Typ 0 und Typ 1 Grammatiken, wie die Länge der Satzformen zunimmt (und evtl. abnimmt), bis das gewünschte Wort der Länge $n$ abgeleitet ist.




*$\varepsilon$-Sonderregelung:* Wegen der Forderung $|w_1| \leq |w_2|$ kann das leere Wort $\varepsilon$ bei Typ 1,2,3 Grammatiken nicht abgeleitet werden, d.h. es gilt immer $\varepsilon \notin L(G)$. Das ist eigentlich nicht wünschenswert. Deshalb soll über die obige Definition hinaus folgende Sonderregelung gelten: Ist $\varepsilon \in L(G)$ erwünscht, so sei die Regel $S \to \varepsilon$ zugelassen ($S$ ist die Startvariable). In diesem Fall ist es dann aber unzulässig, dass $S$ auf der rechten Seite einer Produktion vorkommt. Dies ist keine Beschränkung der Allgemeinheit: Kommt $S$ auf einer rechten Seite vor, so ersetzen wir die alten Regeln (für die Sprache ohne $\varepsilon$) durch folgende (hierbei ist $S'$ eine neue Variable):

1. $S \to$ die rechten Seiten der $S$-Regeln, mit $S$ ersetzt durch $S'$
2. alle Regeln mit $S$ ersetzt durch $S'$
3. $S \to \varepsilon$

Man beachte, dass hierdurch der Typ der Grammatik nicht verändert wird (abgesehen von der nun neu zugelassenen Regel $S \to \varepsilon$).

Bei *kontextfreien* Grammatiken ist es oftmals wünschenswert und bequem, auch Regeln der Form $A \to \varepsilon$ zuzulassen, wobei $A$ nicht unbedingt die Startvariable ist (vgl. das allererste, einführende Beispiel). Für den Typ der kontextfreien und der regulären Grammatik (und nur für diese!) ist nichts dagegen einzuwenden, denn man kann jeder kontextfreien bzw. reguläre Grammatik $G$ mit $\varepsilon \notin L(G)$ eine kontextfreie bzw. reguläre Grammatik $G'$ ohne $\varepsilon$-Regeln zuordnen, die dieselbe Sprache erzeugt. (Falls $\varepsilon \in L(G)$, so kommt noch die obige Sonderregelung hinzu).

Hierzu zerlegen wir die Menge der Variablen $V$ zunächst so in $V_1$ und $V_2$, dass für alle $A \in V_1$ (und nur für diese) gilt $A \Rightarrow^* \varepsilon$. Die Variablenmenge $V_1$ findet man wie folgt: Sofern $A \to \varepsilon$ eine Regel in $P$ ist, so ist $A \in V_1$. Weitere Variablen $B$ in $V_1$ findet man sukzessive dadurch, dass es in $P$ eine Regel $B \to A_1 A_2 \ldots A_k$, $k \geq 1$, gibt mit $A_i \in V_1$ ($i = 1, \ldots, k$). Nach endlich vielen Schritten hat man alle Variablen in $V_1$ gefunden.

Als nächstes entfernen wir alle $\varepsilon$-Regeln aus $P$ und fügen für jede Regel der Form $B \to xAy$ mit $B \in V, A \in V_1, xy \in (V \cup \Sigma)^+$ eine weitere Regel der Form $B \to xy$ zu $P$ hinzu. (Hierdurch wird sozusagen die Möglichkeit, dass $A$ auf das leere Wort abgeleitet werden kann, vorweggenommen). Wenn sich die Regel $B \to \ldots$ in mehrfacher Weise wie oben angegeben zerlegen lässt, so müssen dementsprechend mehrere Regeln der Form $B \to xy$ hinzugefügt werden. Die resultierende Grammatik ist dann $G'$.

Die Typ 3-Sprachen sind echt in der Menge der Typ 2-Sprachen enthalten. Ebenso sind die Typ 2-Sprachen echt in den Typ 1-Sprachen enthalten, und die Typ 1-Sprachen sind echt in den Typ 0-Sprachen enthalten. Beispiele für Sprachen in der jeweiligen Differenzmenge sind:

$$L = \{a^n b^n \mid n \geq 1\}$$

ist vom Typ 2, aber nicht vom Typ 3 (vgl. Seite 78).

$$L' = \{a^n b^n c^n \mid n \geq 1\}$$

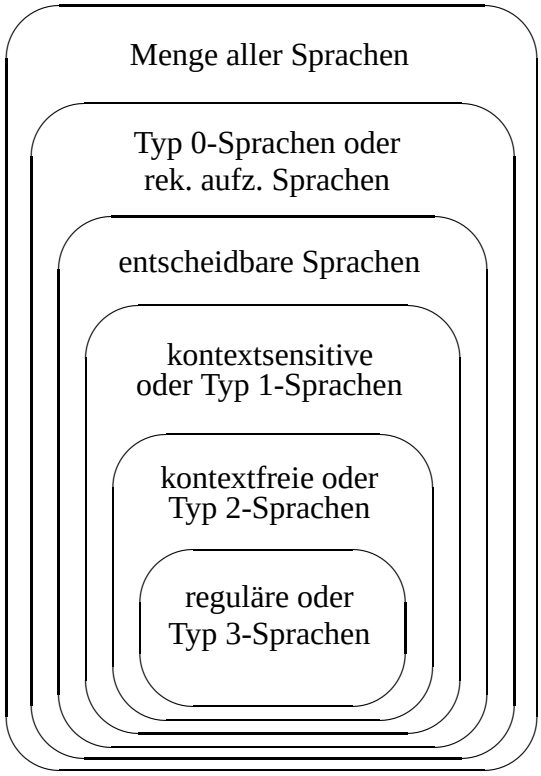
ist vom Typ 1, aber nicht vom Typ 2 (vgl. Seite 93).

$$L'' = H$$

ist vom Typ 0, aber nicht vom Typ 1. (Hierbei ist $H$ das „Halteproblem“). Ferner gibt es entscheidbare Sprachen, die nicht Typ 1 sind. Wir nennen diese aus 4 Sprachklassen bestehende Hierarchie (Typ 3 $\subset$ Typ 2 $\subset$ Typ 1 $\subset$ Typ 0) die *Chomsky-Hierarchie*.

Ferner gilt, dass alle Sprachen vom Typ 1,2,3 *entscheidbar* sind, d.h. es gibt einen Algorithmus, der bei Eingabe von $G$ und $w$ in endlicher Zeit feststellt, ob $w \in L(G)$ oder nicht (der Beweis findet sich weiter unten). Die Typ 0-Sprachen sind identisch mit den *semi-entscheidbaren* oder *rekursiv aufzählbaren Sprachen*. Daher gibt es Typ 0-Sprachen, die nicht entscheidbar sind. (Obige Sprache $L''$ ist ein solches Beispiel).

Da Typ 0-Grammatiken per Definition endliche Objekte sind, ist die Menge aller Typ 0-Grammatiken eine abzählbare Menge, hat also dieselbe Kardinalität wie $\mathbb{N}$, die Menge der natürlichen Zahlen. Da jeder Typ 0-Sprache mindestens eine Typ 0-Grammatik zugeordnet ist, ist die Menge der Typ 0-Sprachen gleichfalls abzählbar. Die Menge *aller* Sprachen aber, sogar schon die Menge aller Teilmengen von $\{0,1\}^*$ ist überabzählbar, sie hat dieselbe Kardinalität wie $\mathbb{R}$, die Menge der reellen Zahlen. Daher muss es Sprachen geben, die nicht durch Grammatiken beschreibbar sind. Mehr noch: maßtheoretisch gesehen hat die Menge der Typ 0-Sprachen das Maß 0. Dass diese Nullmenge trotzdem viel interessante Theorie hergibt, werden wir noch sehen.




Für die praktische Umsetzung in der Informatik (Syntaxanalyse, Compilerbau) sind vor allem die regulären (Typ 3) und kontextfreien (Typ 2) Sprachen von Interesse. Deshalb sind diese Sprach-Typen auch besonders intensiv untersucht worden, und es wurden zwischen Typ 3 und Typ 2 noch weitere Sprachklassen eingebettet (linear kontextfreie Sprachen, deterministisch kontextfreie Sprachen, $LL(k)$- und $LR(k)$-Sprachen, etc.).

Allerdings sind die konkreten Fragestellungen, mit denen man es in der Praxis zu tun hat, im Allgemeinen eher kontextsensitiv (Typ 1) oder sogar Typ 0. Wegen der schwierigeren algorithmischen Handhabung von Typ 0,1-Sprachen, wird trotzdem oft versucht, mit kontextfreien Grammatiken zu arbeiten, und die unterschiedlichen „Kontextbedingungen“ und „Sonderfälle“ dann durch nicht-grammatikalische Zusatzalgorithmen zu behandeln.

Zum Beispiel ist die Menge aller korrekten MODULA-Programme eigentlich nicht kontextfrei (denn Bedingungen wie Typverträglichkeiten, korrekte Anzahl von Parametern in Prozeduraufrufen, ausschließliches Verwenden von vorher deklarierten Objekten, etc. lassen sich nicht durch kontextfreie Grammatiken ausdrücken – wie schon fast der Name „kontextfrei“ suggeriert). Trotzdem werden zur Beschreibung der Syntax von MODULA kontextfreie Grammatiken (bzw. Syntaxdiagramme, BNF) verwendet – mit dem Verständnis, dass über die durch die Syntaxdiagramme festgelegte Syntax hinaus noch Typüberprüfungen etc. zu erfolgen haben.

### 3.1.3 Wortproblem

Sei $S \Rightarrow x_1 \Rightarrow \ldots \Rightarrow x_k = x$ eine Herleitung des Wortes $x$ der Länge $n$ in einer kontext-sensitiven Grammatik. Durch die Bedingung $|w_1| \leq |w_2|$ bei kontext-sensitiven Grammatiken ergibt sich, dass alle „Zwischenergebnisse“, die im Verlauf dieser Herleitung entstehen, höchstens die Länge $n$ haben (vgl. Diagramm auf Seite 59). Da es nur endlich viele Wörter über $(V \cup \Sigma)^*$ der Länge $\leq n$ gibt, ist es einsichtig, dass man durch systematisches Durchprobieren in der Lage ist, in endlicher Zeit zu entscheiden, ob ein gegebenes $x$ in $L(G)$ liegt oder nicht.

Der folgende Satz führt dies präzise aus:

**Satz.**
DAS *Wortproblem* FÜR TYP 1-SPRACHEN (UND DAMIT AUCH FÜR TYP 2, TYP 3-SPRACHEN) IST ENTSCHEIDBAR.
GENAUER:
ES GIBT EINEN ALGORITHMUS, DER BEI EINGABE EINER KONTEXT-SENSITIVEN GRAMMATIK $G = (V, \Sigma, P, S)$ UND EINES WORTES $x \in \Sigma^*$ IN ENDLICHER ZEIT ENTSCHEIDET, OB $x \in L(G)$ ODER $x \notin L(G)$.

**Beweis:** Für $m, n \in I\!N$ definiere Mengen $T_m^n$ wie folgt:

$$T_m^n = \{w \in (V \cup \Sigma)^* \mid |w| \leq n \text{ und } w \text{ lässt sich aus } S \text{ in höchstens } m \text{ Schritten ableiten}\}$$

Die Mengen $T_m^n$, $n \geq 1$, lassen sich induktiv über $m$ definieren:

$$\begin{aligned} T_0^n &= \{S\} \\ T_{m+1}^n &= \mathit{Abl}_n(T_m^n), \text{ wobei} \\ \mathit{Abl}_n(X) &= X \cup \{w \in (V \cup \Sigma)^* \mid |w| \leq n \text{ und } w' \Rightarrow w \text{ für ein } w' \in X\} \end{aligned}$$

Diese Darstellung ist nur für Typ 1-Grammatiken korrekt. (Bei einer Typ 0-Grammatik könnte es ja sein, dass aus einem Wort der Länge $> n$ ein Wort der Länge $\leq n$ ableitbar ist).

Da es nur endlich (nämlich exponentiell in $n$) viele Wörter der Länge $\leq n$ in $(V \cup \Sigma)^*$ gibt, ist

$$\bigcup_{m \geq 0} T_m^n$$

für jedes $n$ eine endliche Menge (der Mächtigkeit $2^{O(n)}$). Daraus ergibt sich, dass es ein $m$ gibt mit

$$T_m^n = T_{m+1}^n = T_{m+2}^n = \ldots$$

Falls nun $x$, $|x| = n$, in $L(G)$ liegt, so muss $x$ in $\bigcup_{m \geq 0} T_m^n$ und damit in $T_m^n$ für ein $m$ liegen. Damit ergibt sich der folgende Algorithmus:

```
INPUT (G, x); { |x| = n }
T := {S};
REPEAT
  T_1 := T;
  T := Abl_n(T_1)
UNTIL (x ∈ T) OR (T = T_1);
IF x ∈ T
  THEN WriteString('x liegt in L(G)')
  ELSE WriteString('x liegt nicht in L(G)')
END
```

□

*Bemerkung:* Der Algorithmus hat exponentielle Laufzeit. Dieses ist durch andere Programmierung auch kaum zu vermeiden, denn das Wortproblem für kontext-sensitive Sprachen ist *NP-hart*.

*Beispiel:* Betrachten wir die obige Beispielgrammatik für $a^n b^n c^n$ mit den Regeln:

$$S \to aSBC, S \to aBC, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc$$

Sei $n = 4$. Dann erhalten wir:

$$\begin{array}{lcl} T_0^4 & = & \{S\} \\ T_1^4 & = & \{S, aSBC, aBC\} \\ T_2^4 & = & \{S, aSBC, aBC, abC\} \\ T_3^4 & = & \{S, aSBC, aBC, abC, abc\} \\ T_4^4 & = & \{S, aSBC, aBC, abC, abc\} \; = \; T_3^4 \end{array}$$

Das heißt, das einzige Wort der Sprache $L(G)$ der Länge $\leq 4$ ist $abc$ (wie wir bereits wissen).

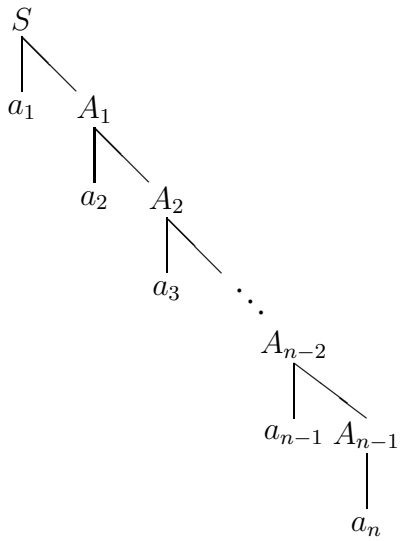
### 3.1.4 Syntaxbäume

Wir haben die Syntaxbäume informell an den Beispielen schon eingeführt. Wir wollen dies nun etwas formaler machen.

Einer Ableitung eines Wortes $x$ in einer Typ 2 (oder 3) Grammatik $G$ kann man einen Syntaxbaum oder Ableitungsbaum zuordnen. Sei $x \in L(G)$ und sei $S = x_0 \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow x_n = x$ eine Ableitung des Wortes $x$. Dann wird der Wurzel des Syntaxbaumes die Startvariable $S$ zugeordnet. Für $i = 1, 2, \ldots, n$ gehe man nun wie folgt vor: Falls im $i$-ten Ableitungsschritt (also beim Übergang von $x_{i-1}$ nach $x_i$) gerade die Variable $A$ durch ein Wort $z$ ersetzt wird (wegen $A \to z \in P$), dann sehe im Syntaxbaum $|z|$ viele

Söhne von $A$ vor und beschrifte diese mit den einzelnen Zeichen von $z$. Auf diese Weise entsteht ein Baum, dessen Blätter gerade mit den Symbolen in $x$ beschriftet sind.

Man beachte, dass Ableitungsbäume bei regulären Grammatiken immer die „entartete“ Form einer nach rechts geneigten linearen Kette haben:



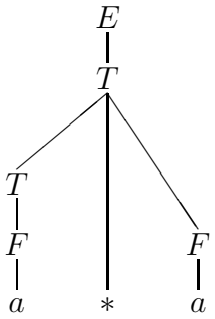

Verschiedenen Ableitungen kann derselbe Syntaxbaum zugeordnet sein: Die folgenden beiden Ableitungen von $a * a$ (vgl. obige Beispielgrammatik)

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow a * F \Rightarrow a * a$$

und

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * a \Rightarrow F * a \Rightarrow a * a$$

besitzen denselben Syntaxbaum:




Die erste der beiden Ableitungen zeichnet sich dadurch aus, dass in jedem Schritt immer die erste vorkommende – also am weitesten links stehende – Variable durch eine Regelanwendung ersetzt wird. Dies ist eine sog. *Linksableitung*. Man kann umgekehrt offensichtlich jedem gegebenen Syntaxbaum eindeutig eine Linksableitung zuordnen. Daher erhalten wir:

$$x \in L(G) \quad \Leftrightarrow \quad \text{es gibt (irgend)eine Ableitung von } x$$

$\Leftrightarrow$ es gibt einen Syntaxbaum mit $x$ an den Blättern

$\Leftrightarrow$ es gibt eine Linksableitung von $x$

Entsprechendes gilt natürlich auch für die *Rechtsableitung*.

Trotzdem kann es vorkommen, dass es für dasselbe Wort $x$ verschiedenartig strukturierte Syntaxbäume gibt. Man sagt dann, dass die Grammatik *mehrdeutig* ist. Wenn für kein Wort $x$ dieser Fall eintritt, heißt die Grammatik *eindeutig*. In der Grammatik

$$\begin{aligned} S &\rightarrow aB \\ S &\rightarrow Ac \\ A &\rightarrow ab \\ B &\rightarrow bc \end{aligned}$$

gibt es zwei unterschiedliche Syntaxbäume für das Wort $abc$:


In diesem Fall kann natürlich die Mehrdeutigkeit beseitigt werden: man kann eine andere, eindeutige Grammatik angeben, die dieselbe Sprache definiert (z.B.: $S \rightarrow abc$). Es gibt aber Fälle, bei denen die Mehrdeutigkeit unvermeidbar ist. Eine kontextfreie Sprache $A$ heißt *inhärent mehrdeutig*, wenn *jede* Grammatik $G$ mit $L(G) = A$ mehrdeutig ist.

Ein Beispiel für eine inhärent mehrdeutige, kontextfreie Sprache ist

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

### 3.1.5 Backus-Naur-Form

Von Backus und Naur stammt ein Formalismus zum kompakten Niederschreiben von kontexfreien, also Typ 2, Grammatiken, kurz BNF genannt. (Diese Notation wurde im Zusammenhang mit der Konstruktion der Programmiersprache ALGOL 60 eingeführt).

Und zwar schreiben wir bei mehreren Regeln, die alle dieselbe linke Seite haben

$$\begin{aligned} A &\rightarrow \beta_1 \\ A &\rightarrow \beta_2 \\ &\vdots \\ A &\rightarrow \beta_n \end{aligned}$$

kürzer nur eine einzige „Metaregel“ (unter Verwenden des „Metasymbols“ |):

$$A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

(Backus und Naur verwenden statt $\rightarrow$ allerdings $::=$).

Wir werden im Folgenden auch diese abkürzende Notation verwenden.

Der von Backus und Naur verwendete Formalismus geht allerdings noch weiter (man spricht dann von *erweiterter BNF*, kurz: EBNF).

So steht z.B.

$$A \rightarrow \alpha[\beta]\gamma$$

für die Regeln

$$\begin{aligned} A &\rightarrow \alpha\gamma \\ A &\rightarrow \alpha\beta\gamma \end{aligned}$$

*Bedeutung:* Das Wort $\beta$ kann – muss aber nicht – zwischen $\alpha$ und $\gamma$ eingefügt werden.

Ferner steht

$$A \rightarrow \alpha\{\beta\}\gamma$$

für die Regeln

$$\begin{aligned} A &\rightarrow \alpha\gamma \\ A &\rightarrow \alpha B\gamma \\ B &\rightarrow \beta \\ B &\rightarrow \beta B \end{aligned}$$

*Bedeutung:* Das Wort $\beta$ kann zwischen $\alpha$ und $\gamma$ beliebig oft (auch null-mal) wiederholt werden.

Da (E)BNF und kontextfreie Grammatiken gleichwertig sind, heißt das also, dass durch (E)BNF exakt die kontextfreien, also Typ 2 Sprachen dargestellt werden.

## 3.2 Reguläre Sprachen

In diesem Abschnitt wollen wir die regulären (also Typ 3) Sprachen näher untersuchen. Wir werden verschiedene äquivalente Charakterisierungen beweisen (z.B. mittels (nicht)deterministischer endlicher Automaten, mittels regulärer Ausdrücke und mittels gewisser Äquivalenzrelationen) und Eigenschaften der regulären Sprachen angeben (Pumping Lemma, Abschlusseigenschaften).

### 3.2.1 Endliche Automaten

Ein (deterministischer, endlicher) Automat wird – bildhaft gesprochen – auf ein Eingabewort angesetzt und dieser „erkennt“ (oder „akzeptiert“) dieses Wort schließlich – oder auch nicht. Die Menge der akzeptierten Wörter bildet dann die durch den Automaten dargestellte oder definierte Sprache. Bei den Grammatiken war der Mechanismus in gewisser Weise umgekehrt: Die Grammatik „erzeugt“ durch entsprechende Regelanwendungen ein Wort. Das Wort der Sprache entsteht also erst am Ende des Erzeugungsprozesses.

Wir beginnen mit der Definition der *endlichen Automaten* (englisch: deterministic finite automaton, kurz: DFA).

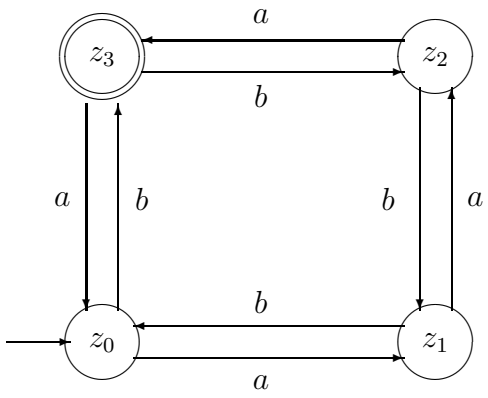
> **Definition.** Ein *(deterministischer) endlicher Automat* $M$ wird spezifiziert durch ein 5-Tupel
>
> $$M = (Z, \Sigma, \delta, z_0, E).$$
>
> Hierbei bezeichnet $Z$ die Menge der *Zustände* und $\Sigma$ ist das *Eingabealphabet*, $Z \cap \Sigma = \emptyset$. $Z$ und $\Sigma$ müssen – wie schon der Name sagt – endliche Mengen sein. $z_0 \in Z$ ist der *Startzustand*, $E \subseteq Z$ ist die Menge der *Endzustände* und $\delta : Z \times \Sigma \longrightarrow Z$ heißt die *Überführungsfunktion*.

Wir veranschaulichen uns endliche Automaten durch ihren *Zustandsgraphen*, der ein gerichteter, beschrifteter Graph ist, wobei die Zustände die Knoten sind. Der Knoten, der dem Startzustand entspricht, wird durch einen hineingehenden Pfeil besonders markiert und alle Endzustände werden durch doppelte Kreise gekennzeichnet. Die Kanten in dem Graphen sind folgendermaßen definiert: Von $z_1$ nach $z_2$ geht eine mit $a \in \Sigma$ beschriftete Kante, falls $\delta(z_1, a) = z_2$.

*Beispiel:* Sei $M = (Z, \Sigma, \delta, z_0, E)$, wobei

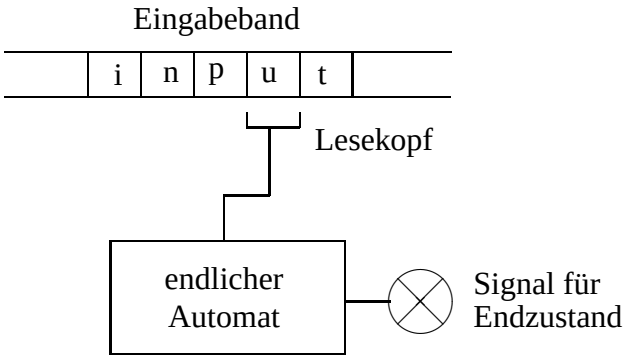
$$\begin{aligned} Z &= \{z_0, z_1, z_2, z_3\} \\ \Sigma &= \{a, b\} \\ E &= \{z_3\} \\ \delta(z_0, a) &= z_1 \\ \delta(z_0, b) &= z_3 \\ \delta(z_1, a) &= z_2 \\ \delta(z_1, b) &= z_0 \\ \delta(z_2, a) &= z_3 \\ \delta(z_2, b) &= z_1 \\ \delta(z_3, a) &= z_0 \\ \delta(z_3, b) &= z_2 \end{aligned}$$

Der zuständige Graph sieht folgendermaßen aus:




Ein endlicher Automat beschreibt (erkennt, akzeptiert) eine Sprache $L \subseteq \Sigma^*$ wie folgt. Ein einzelnes Wort $a_1 a_2 \ldots a_n$ wird von dem Automaten dadurch „erkannt“, dass der Automat beim „Lesen“ dieses Eingabeworts eine Folge von Zuständen $z_0, z_1, \ldots, z_n$ durchläuft. Hierbei ist $z_0$ der Startzustand und es gilt $\delta(z_{i-1}, a_i) = z_i$ für $i = 1, \ldots, n$. Und schließlich muss $z_n$ in $E$ liegen, also ein Endzustand sein.

*Bildhafte Deutung:*




Der obige Beispielautomat erkennt somit die Sprache

$$L = \{x \in \Sigma^* \mid ((\text{Anzahl } a\text{'s in } x) - (\text{Anzahl } b\text{'s in } x)) \equiv 3 \pmod 4\}$$

Die folgende Definition führt diese informale Erklärung formal aus.

**Definition.** Zu einem gegebenen DFA $M = (Z, \Sigma, \delta, z_0, E)$ definieren wir eine Funktion $\hat{\delta} : Z \times \Sigma^* \longrightarrow Z$ durch eine induktive Definition wie folgt:

$$\begin{aligned} \hat{\delta}(z, \varepsilon) &= z \\ \hat{\delta}(z, ax) &= \hat{\delta}(\delta(z, a), x) \end{aligned}$$

Hierbei ist $z \in Z$, $x \in \Sigma^*$ und $a \in \Sigma$.

Die von $M$ *akzeptierte Sprache* ist

$$T(M) = \{x \in \Sigma^* \mid \hat{\delta}(z_0, x) \in E\}$$

Die Definition von $\hat{\delta}$ erweitert offensichtlich die Definition von $\delta$ von Einzelzeichen zu Wörtern. Für ein einzelnes Zeichen $a$ gilt: $\hat{\delta}(z, a) = \delta(z, a)$. Es gilt ferner:

$$\hat{\delta}(z, a_1 a_2 \dots a_n) = \delta(\dots \delta(\delta(z, a_1), a_2) \dots, a_n)$$

**Satz.**
JEDE DURCH ENDLICHE AUTOMATEN ERKENNBARE SPRACHE IST REGULÄR (ALSO TYP 3).

**Beweis:** Sei $A \subseteq \Sigma^*$ eine Sprache und $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA mit $T(M) = A$. Wir definieren nun eine Typ 3 Grammatik $G$ mit $L(G) = A$. Es ist $G = (V, \Sigma, P, S)$, wobei $V = Z$ und $S = z_0$. Sofern $\varepsilon \in T(M)$ (d.h. falls $z_0 \in E$), so enthält $P$ die Regel $z_0 \to \varepsilon$ (was evtl. weitere Umstrukturierungen nach sich zieht, vgl. S. 59). Ferner besteht $P$ aus den folgenden Regeln: Jeder $\delta$-„Anweisung“

$$\delta(z_1, a) = z_2$$

ordnen wir folgende Regel(n) in $P$ zu:

$$z_1 \to a z_2 \ \in \ P$$

und zusätzlich, falls $z_2 \in E$,

$$z_1 \to a \ \in \ P.$$

Nun gilt für alle $x = a_1 a_2 \dots a_n \in \Sigma^*$:

$x \in T(M)$

**genau dann wenn** es gibt eine Folge von Zuständen $z_0, z_1, \dots, z_n$ mit: $z_0$ ist Startzustand, $z_n \in E$ und für $i = 1, \dots, n$: $\delta(z_{i-1}, a_i) = z_i$

**genau dann wenn** es gibt eine Folge von Variablen $z_0, z_1, \dots, z_{n-1}$ mit: $z_0$ ist Startvariable und es gilt: $z_0 \Rightarrow a_1 z_1 \Rightarrow a_1 a_2 z_2 \Rightarrow \dots \Rightarrow a_1 a_2 \dots a_{n-1} z_{n-1} \Rightarrow a_1 a_2 \dots a_{n-1} a_n$

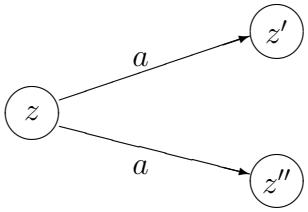
**genau dann wenn** $x \in L(G)$. □

### 3.2.2 Nichtdeterministische Automaten

Tatsächlich gilt auch die Umkehrung des Satzes, so dass also eine Sprache regulär ist *genau dann, wenn* sie von einem endlichen Automaten erkannt wird. Um diese Richtung (reguläre Grammatik $\longrightarrow$ DFA) zu beweisen, benötigen wir ein beweistechnisches

Hilfsmittel, sozusagen als Zwischenschritt, den *nichtdeterministischen Automaten* (engl.: nondeterministic finite automaton, kurz: NFA).

Wir erläutern den NFA zunächst informal. Bei einem NFA ist es zugelassen, dass vom selben Zustand $z \in Z$ aus mehrere (oder auch gar keine) Pfeile ausgehen, die mit $a \in \Sigma$ beschriftet sind.




Wie soll nun ein derartiger Automat „funktionieren“? Wenn im Zustand $z$ das Zeichen $a$ gelesen wird, so *kann* der Automat sowohl in $z'$ als auch in $z''$ übergehen. Beide Möglichkeiten werden gleichwertig behandelt. Ein Wort $x \in \Sigma^*$ gilt dann schon vom Automaten akzeptiert, wenn er bei Lesen von $x$ eine Zustandsfolge durchlaufen *kann*, die auf einen Endzustand führt. Andere mögliche Zustandsfolgen können durchaus auf Nicht-Endzustände führen. Wichtig ist, dass es *mindestens eine* akzeptierende Zustandsfolge gibt. Konsequenterweise lassen wir nun auch eine *Menge* von Startzuständen, anstatt eines einzigen, zu.

Die folgende Definition formalisiert dies.

**Definition:** Ein nichtdeterministischer, endlicher Automat (kurz: NFA) wird spezifiziert durch ein 5-Tupel

$$M = (Z, \Sigma, \delta, S, E)$$

Hierbei sind:

- $Z$ eine endliche Menge, die *Zustände*
- $\Sigma$ eine endliche Menge, das *Eingabealphabet*, $Z \cap \Sigma = \emptyset$
- $\delta$ eine Funktion von $Z \times \Sigma$ nach $\mathcal{P}(Z)$, die *Überführungsfunktion*
- $S \subseteq Z$ die Menge der *Startzustände*
- $E \subseteq Z$ die Menge der *Endzustände*

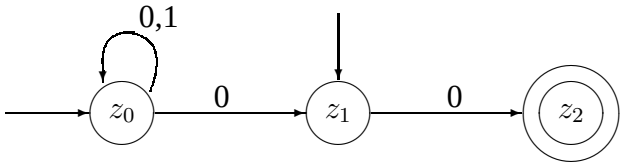
Die Funktion $\delta$ kann wieder verallgemeinert werden zu $\hat{\delta} : \mathcal{P}(Z) \times \Sigma^* \longrightarrow \mathcal{P}(Z)$, und zwar durch folgende induktive Definition:

$$\begin{aligned} \hat{\delta}(Z', \varepsilon) &= Z' \text{ für alle } Z' \subseteq Z \\ \hat{\delta}(Z', ax) &= \bigcup_{z \in Z'} \hat{\delta}(\delta(z, a), x) \end{aligned}$$

Die von einem NFA *akzeptierte Sprache* ist

$$T(M) = \{x \in \Sigma^* \mid \hat{\delta}(S, x) \cap E \neq \emptyset\}$$

*Beispiel:* Der folgende NFA akzeptiert genau die Wörter $x$ über $\{0,1\}$, die mit 00 enden, oder falls $x = 0$ ist.




**Satz (Rabin, Scott).**
JEDE VON EINEM NFA AKZEPTIERBARE SPRACHE IST AUCH DURCH EINEN DFA AKZEPTIERBAR.

**Beweis:** Sei $M = (Z, \Sigma, \delta, S, E)$ ein gegebener NFA. Wir konstruieren einen DFA $M'$, der ebenfalls $T(M)$ akzeptiert, dadurch dass wir in $M'$ jede mögliche Teilmenge der Zustände von $M$ (also die Elemente der Potenzmenge von $Z$) als *Einzelzustand* von $M'$ vorsehen. Die restlichen Teile der Definition von $M'$ ergeben sich dann mehr oder weniger zwangsläufig.

Wir setzen also

$$M' = (\mathcal{Z}, \Sigma, \delta', z_0', E')$$

wobei

$$\begin{aligned} \mathcal{Z} &= \mathcal{P}(Z) \\ \delta'(Z', a) &= \bigcup_{z \in Z'} \delta(z, a) \;=\; \hat{\delta}(Z', a),\ Z' \in \mathcal{Z} \\ z_0' &= S \\ E' &= \{Z' \subseteq Z \mid Z' \cap E \neq \emptyset\} \end{aligned}$$

Es ist klar, dass nun für alle $x = a_1 \ldots a_n \in \Sigma^*$ gilt:

$x \in T(M)$

**genau dann wenn** $\hat{\delta}(S, x) \cap E \neq \emptyset$

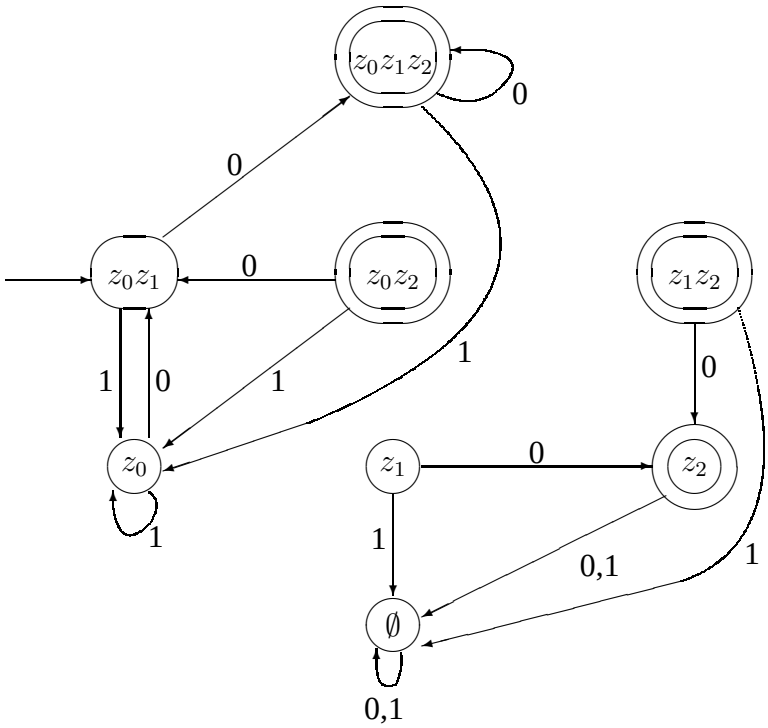
**genau dann wenn** es gibt eine Folge von Teilmengen $Z_1, Z_2, \ldots, Z_n$ von $Z$ mit $\delta'(S, a_1) = Z_1$, $\delta'(Z_1, a_2) = Z_2$, …, $\delta'(Z_{n-1}, a_n) = Z_n$ und $Z_n \cap E \neq \emptyset$.

**genau dann wenn** $\hat{\delta}'(S, x) \in E'$

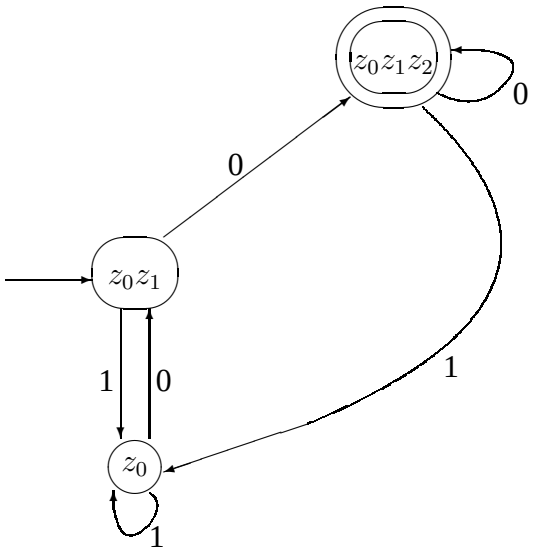
**genau dann wenn** $x \in T(M')$. □

*Beispiel:* Für obigen nichtdeterministischen Automaten ergibt sich aus dem Beweis der folgende deterministische Automat mit den 8 Zuständen $\emptyset$, $\{z_0\}$, $\{z_1\}$, $\{z_2\}$, $\{z_0, z_1\}$, $\{z_0, z_2\}$, $\{z_1, z_2\}$, $\{z_0, z_1, z_2\}$. Der Startzustand ist $\{z_0, z_1\}$ (da $z_0$ und $z_1$ die Startzustände

des NFAs sind). Die Endzustände des neuen Automaten sind alle Zustände, die mindestens einen ursprünglichen Endzustand enthalten.




Im allgemeinen enthält der so entstandene deterministische Automat viele überflüssige Zustände. Wenn wir alle Zustände entfernen, die vom Startzustand (hier: $z_0z_1$) nicht erreichbar sind, erhalten wir:

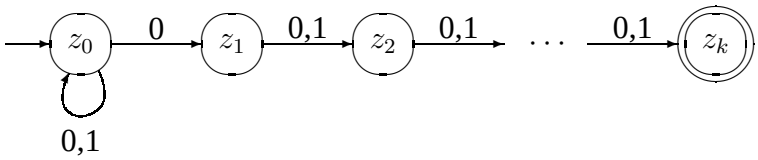



Auch dieser Automat muss noch nicht minimal sein. (In diesem Fall ist er es allerdings).

*Beispiel:* Die folgende Sprache

$$L_k \;=\; \{x \in \{0,1\}^* \mid |x| \geq k, \text{ das } k\text{-letzte Zeichen von } x \text{ ist } 0\}, \quad k \geq 1$$

kann durch einen NFA mit $k+1$ Zuständen erkannt werden:



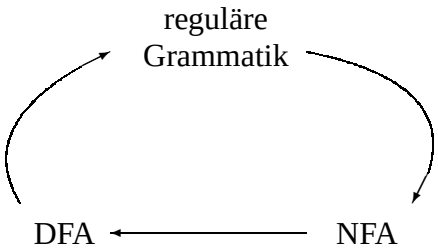

Wir behaupten, dass es keinen DFA gibt, der die Sprache $L_k$ mit weniger als $2^k$ vielen Zuständen akzeptieren kann. Dieses Beispiel zeigt also, dass die Umformung von NFA nach DFA im Allgemeinen auf einen DFA führt, der nicht allzu viel verbessert werden kann. Außerdem wird durch dieses Beispiel gezeigt, dass NFA's reguläre Sprachen unter Umständen viel kompakter repräsentieren können als DFA's.

Wenn $M$ ein DFA mit $< 2^k$ Zuständen ist, dann muss es zwei Wörter $y_1, y_2 \in \{0,1\}^k$ geben, so dass $\hat{\delta}(z_0, y_1) = \hat{\delta}(z_0, y_2)$. Sei $i$ die erste Position, an der sich $y_1$ und $y_2$ unterscheiden ($1 \leq i \leq k$). Sei $w \in \{0,1\}^{i-1}$ beliebig. Dann gilt (o.B.d.A): $y_1 w = u0vw$ und $y_2 w = u1v'w$, wobei $|v| = |v'| = k - i$ und $|u| = i - 1$. Die Null (bzw. die Eins) in $y_1 w$ (bzw. $y_2 w$) kommt an der $k$-letzten Stelle vor. Daher ist $y_1 w \in L_k$ und $y_2 w \notin L_k$. Andererseits gilt:

$$\begin{aligned} \hat{\delta}(z_0, y_1 w) &= \hat{\delta}(\hat{\delta}(z_0, y_1), w) \\ &= \hat{\delta}(\hat{\delta}(z_0, y_2), w) \\ &= \hat{\delta}(z_0, y_2 w) \end{aligned}$$

Daher gilt $y_1 w \in L_k \Leftrightarrow y_2 w \in L_k$. Dieser Widerspruch zeigt, dass $M$ die Sprache $L_k$ nicht erkennen kann.

Mit dem nächsten Satz zeigen wir schließlich, dass die durch (deterministische oder nichtdeterministische) endliche Automaten erkennbaren Sprachen genau die regulären Sprachen sind. Der Satz schließt noch die letzte Lücke in folgendem Ringschluss:




**Satz.**
FÜR JEDE REGULÄRE GRAMMATIK $G$ GIBT ES EINEN NFA $M$ MIT $L(G) = T(M)$.

**Beweis:** Sei die reguläre Grammatik $G = (V, \Sigma, P, S)$ gegeben. Wir geben einen geforderten NFA wie folgt an: $M = (Z, \Sigma, \delta, S', E)$ wobei

$$\begin{aligned} Z &= V \cup \{X\}, \quad X \notin V \\ S' &= \{S\} \end{aligned}$$

$$
\begin{aligned}
E &= \begin{cases} \{S, X\}, & S \to \varepsilon \in P \\ \{X\}, & S \to \varepsilon \notin P \end{cases} \\
\delta(A, a) &\ni B \text{ falls } A \to aB \in P \\
\delta(A, a) &\ni X \text{ falls } A \to a \in P
\end{aligned}
$$

Nun gilt (für $n \geq 1$):

$a_1 a_2 \ldots a_n \in L(G)$

**genau dann wenn** es gibt eine Folge von Variablen $A_1, \ldots, A_{n-1}$ mit: $S \Rightarrow a_1 A_1 \Rightarrow a_1 a_2 A_2 \Rightarrow \cdots \Rightarrow a_1 a_2 \ldots a_{n-1} A_{n-1} \Rightarrow a_1 a_2 \ldots a_{n-1} a_n$

**genau dann wenn** es gibt eine Folge von Zuständen $A_1, \ldots, A_{n-1}$ mit: $\delta(S, a_1) \ni A_1$, $\delta(A_1, a_2) \ni A_2$, …, $\delta(A_{n-1}, a_n) \ni X$

**genau dann wenn** $a_1 a_2 \ldots a_n \in T(M)$. □

Eine Konsequenz dieser Äquivalenzen ist, dass es keine regulären Sprachen gibt, die inhärent mehrdeutig sind. Mit anderen Worten: für jede reguläre Sprache gibt es eine eindeutige reguläre Grammatik, die diese Sprache erzeugt. Denn, wenn $L$ eine reguläre Sprache ist, so kann man zu $L$ einen DFA angeben, der $L$ akzeptiert. Die Umformung vom DFA zu einer regulären Grammatik erzeugt eine eindeutige Grammatik.

### 3.2.3 Reguläre Ausdrücke

Reguläre Ausdrücke sind spezielle Formeln, mit denen Sprachen definiert werden können. (Wie wir später sehen werden, sind dies wieder genau die regulären Sprachen). Diese Ausdrücke werden wie folgt induktiv definiert:

- $\emptyset$ ist ein regulärer Ausdruck.
- $\varepsilon$ ist ein regulärer Ausdruck.
- Für jedes $a \in \Sigma$ ist $a$ ist ein regulärer Ausdruck.
- Wenn $\alpha$ und $\beta$ reguläre Ausdrücke sind, dann auch $\alpha\beta$, $(\alpha|\beta)$, sowie $(\alpha)^*$.

Einem regulären Ausdruck $\gamma$ wird in eindeutiger Weise – induktiv über den Aufbau von $\gamma$ – eine Sprache zugeordnet, die wir mit $L(\gamma)$ bezeichnen.
Es gilt:

- Falls $\gamma = \emptyset$, so ist $L(\gamma) = \emptyset$.
- Falls $\gamma = \varepsilon$, so ist $L(\gamma) = \{\varepsilon\}$.

- Falls $\gamma = a$, so ist $L(\gamma) = \{a\}$.
- Falls $\gamma = \alpha\beta$, so ist $L(\gamma) = L(\alpha)L(\beta)$ (das Produkt von $L(\alpha)$ und $L(\beta)$).
- Falls $\gamma = (\alpha|\beta)$, so ist $L(\gamma) = L(\alpha) \cup L(\beta)$.
- Falls $\gamma = (\alpha)^*$, so ist $L(\gamma) = L(\alpha)^*$.

*Beispiel:* Durch den regulären Ausdruck

$$(0|(0|1)^*00)$$

wird die weiter oben diskutierte Beispielsprache beschrieben. (Entweder $x = 0$ oder $x$ endet mit $00$).

*Bemerkung:* Alle *endlichen* Sprachen sind durch reguläre Ausdrücke beschreibbar, denn sei $A = \{x_1, x_2, \ldots, x_k\}$ gegeben, so ist $\alpha = (\ldots((x_1|x_2)|x_3)\ldots|x_k)$ ein regulärer Ausdruck für $A$, also $L(\alpha) = A$.

**Satz (Kleene).**
DIE MENGE DER DURCH REGULÄRE AUSDRÜCKE BESCHREIBBAREN SPRACHEN IST GENAU DIE MENGE DER REGULÄREN SPRACHEN.

**Beweis:** ($\Rightarrow$) Wir zeigen zunächst: Wenn $L$ durch einen regulären Ausdruck beschreibbar ist, dann gibt es auch einen NFA für $L$. Sei also $L = L(\gamma)$, wobei $\gamma$ ein regulärer Ausdruck ist. In den Fällen $\gamma = \emptyset$, $\gamma = \varepsilon$ und $\gamma = a$ ist es klar, dass $L$ durch einen DFA oder NFA beschreibbar ist.

Habe nun $\gamma$ die Form $\gamma = \alpha\beta$ und seien $M_1$, $M_2$ die nach Induktionsvoraussetzung existierenden NFAs für $L(\alpha)$ und $L(\beta)$. Wir schalten nun die beiden Automaten „in Serie“ wie folgt und erhalten einen NFA $M$ für $L$. Dieser Automat $M$ hat dieselben Startzustände wie $M_1$ und genau die Endzustände von $M_2$. (Falls $\varepsilon \in L(\alpha) = T(M_1)$, so sind auch die Startzustände von $M_2$ Startzustände von $M$). Alle Zustände in $M_1$, die einen Pfeil zu einem Endzustand von $M_1$ haben, erhalten zusätzlich genauso beschriftete Pfeile zu den Startzuständen von $M_2$. Es ist klar, dass $T(M) = T(M_1)T(M_2) = L(\alpha\beta)$.

Wenn $\gamma$ die Form hat $\gamma = (\alpha|\beta)$ und $M_1 = (Z_1, \Sigma, \delta_1, S_1, E_1)$, $M_2 = (Z_2, \Sigma, \delta_2, S_2, E_2)$, $Z_1 \cap Z_2 = \emptyset$, entsprechende NFAs für $L(\alpha)$ und $L(\beta)$ sind, so bilden wir einfach den „Vereinigungsautomaten“

$$M = (Z_1 \cup Z_2, \Sigma, \delta_1 \cup \delta_2, S_1 \cup S_2, E_1 \cup E_2)$$

der offensichtlich die Sprache $L$ akzeptiert.

Falls $\gamma = (\alpha)^*$, so bilden wir aus dem NFA $M$ für $\alpha$ einen Automaten $M'$ für $\gamma$ wie folgt.

Falls $\varepsilon \notin T(M)$, so sehe zunächst einen zusätzlichen Zustand vor, der zugleich Start- und Endzustand ist, der keine weitere Verbindung mit dem Rest des Automaten hat. Dieser modifizierte Automat erkennt nun $\{\varepsilon\} \cup L(\alpha)$.

$M'$ entsteht aus (dem evtl. modifizierten) $M$ wie folgt: $M'$ hat dieselben Startzustände, sowie dieselben Endzustände. Ferner erhält jeder Zustandsknoten, der eine (mit $a$ beschriftete) Verbindung zu einem der ursprünglichen Endzustände hat, zusätzlich einen mit $a$ beschrifteten Pfeil zu jedem Startzustand.

Durch diese „Rückkopplung“ und das Hinzufügen von $\varepsilon$ ist klar, dass gilt: $T(M') = T(M)^* = L((\alpha)^*)$.

($\Leftarrow$) Wir gehen nun von einem DFA $M$ aus und geben einen regulären Ausdruck $\gamma$ an mit $L(\gamma) = T(M)$. Wir nehmen hierzu an, die Zustände von $M$ sind von 1 an durchnummeriert: $Z = \{z_1, \ldots, z_n\}$, so dass $z_1$ der Startzustand ist. Für $i, j \in \{1, \ldots, n\}$ und $k \in \{0, 1, \ldots, n\}$ definieren wir nun Sprachen $R^k_{i,j}$ und zeigen, dass diese durch reguläre Ausdrücke beschreibbar sind.

Es ist

> $R^k_{i,j} = \{x \in \Sigma^* \mid$ die Eingabe $x$ überführt den Automaten, gestartet im Zustand $z_i$ in den Zustand $z_j$ (also $\hat{\delta}(z_i, x) = z_j$), so dass keiner der „Zwischenzustände“ – außer $z_i$ und $z_j$ selbst – einen Index größer als $k$ hat $\}$

Für $k = 0$ und $i \neq j$ gilt:

$$R^0_{i,j} = \{a \in \Sigma \mid \delta(z_i, a) = z_j\}$$

Für $k = 0$ und $i = j$ gilt:

$$R^0_{i,i} = \{\varepsilon\} \cup \{a \in \Sigma \mid \delta(z_i, a) = z_i\}$$

In diesen Fällen ist $R^k_{i,j}$ endlich und lässt sich daher durch einen regulären Ausdruck beschreiben. Wir fahren nun mit einer Induktion über $k$ fort. Wir beobachten zunächst, dass gilt:

$$R^{k+1}_{i,j} = R^k_{i,j} \cup R^k_{i,k+1}(R^k_{k+1,k+1})^* R^k_{k+1,j}$$

(Erklärung: Um vom Zustand $z_i$ aus den Zustand $z_j$ zu erreichen, wird entweder der Zwischenzustand $z_{k+1}$ nicht benötigt, dann reicht $R^k_{i,j}$ zur Beschreibung aus; oder der Zustand $z_{k+1}$ wird ein oder mehrfach (in Schleifen) durchlaufen. Dies kann beschrieben werden durch den Ausdruck $R^k_{i,k+1}(R^k_{k+1,k+1})^* R^k_{k+1,j}$.)

Falls $\alpha^k_{i,j}$ ein regulärer Ausdruck für $R^k_{i,j}$ ist, so lässt sich die obige induktive Formel wie folgt schreiben:

$$\alpha^{k+1}_{i,j} = (\alpha^k_{i,j} | \alpha^k_{i,k+1}(\alpha^k_{k+1,k+1})^* \alpha^k_{k+1,j})$$

Mittels dieser regulären Ausdrücke lässt sich die von $M$ akzeptierte Sprache leicht beschreiben, denn es gilt:

$$T(M) = \bigcup_{z_i \in E} R^n_{1,i}$$

Seien also $i_1, i_2, \ldots, i_m$ die Indizes der Endzustände, so ist ein regulärer Ausdruck für $T(M)$ gegeben durch:

$$(\alpha^n_{1,i_1} | \alpha^n_{1,i_2} | \ldots | \alpha^n_{1,i_m})$$

□

### 3.2.4 Das Pumping Lemma

Wir zeigen nun einen wichtigen Satz, der das Haupthilfsmittel darstellt, um von einer Sprache nachzuweisen, dass sie *nicht* regulär ist.

**Satz (Pumping Lemma, Schleifenlemma, Iterationslemma, Lemma von Bar-Hillel, $uvw$-Theorem).**
SEI $L$ EINE REGULÄRE SPRACHE. DANN GIBT ES EINE ZAHL $n_0$, SO DASS SICH ALLE WÖRTER $x \in L$ MIT $|x| \geq n_0$ ZERLEGEN LASSEN IN $x = uvw$, SO DASS FOLGENDE EIGENSCHAFTEN ERFÜLLT SIND:

1. $|v| \geq 1$,
2. $|uv| \leq n_0$,
3. FÜR ALLE $i = 0, 1, 2, \ldots$ GILT: $uv^i w \in L$.

**Beweis:** Da $L$ regulär ist, gibt es einen DFA $M$, der $L$ akzeptiert.

Wir wählen für $n_0$ die Zahl der Zustände von $M$: $n_0 = |Z|$. Sei nun $x$ ein beliebiges Wort der Länge $\geq n_0$, das der Automat akzeptiert. Beim Abarbeiten von $x$ durchläuft der Automat $|x| + 1$ Zustände (den Startzustand mitgezählt). Da $|x| \geq n_0$ können diese Zustände nicht alle verschieden sein (Schubfachschluss). Mit anderen Worten, der DFA muss beim Abarbeiten von $x$ eine Schleife durchlaufen haben. Wir wählen die Zerlegung $x = uvw$ so, dass der Zustand nach Lesen von $u$ und von $uv$ derselbe ist. Es ist klar, dass die Zerlegung so gewählt werden kann, dass die Bedingungen 1 und 2 erfüllt sind. Da die Zustände gleich sind, erreicht der Automat bei Eingabe von $uw$ denselben Endzustand wie bei Lesen von $x = uvw$. Das heißt $uw = uv^0w \in L$. Dasselbe gilt für $uvvw = uv^2w$, $uvvvw = uv^3w$, usw. Daher ist auch die Bedingung 3 erfüllt. □

Man beachte, dass durch das Pumping Lemma nicht eine äquivalente Charakterisierung der regulären Sprachen erreicht wird. Es stellt lediglich eine einseitige Implikation dar. Die logische Struktur ist

$$(L \text{ regulär}) \Rightarrow (\exists n_0)(\forall z \in L, |z| \geq n_0)(\exists u, v, w)[(z = uvw) \wedge 1 \wedge 2 \wedge 3]$$

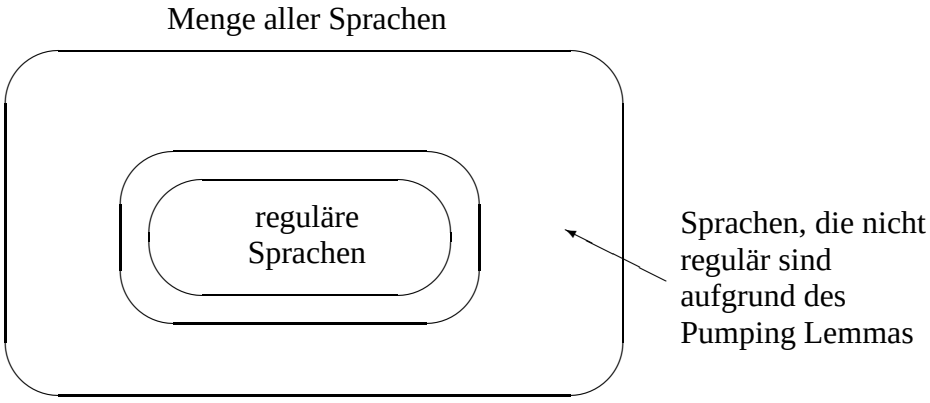
Daher ist die typische Anwendung des Pumping Lemmas der Nachweis, dass gewisse Sprachen *nicht* regulär sind. Dies muss allerdings nicht bei jeder nicht-regulären Sprache gelingen! Die folgende nicht-reguläre Sprache über $\{a, b, c\}$ ist ein Beispiel hierfür

$$L = \{c^m a^n b^n \mid m, n \geq 0\} \cup \{a, b\}^*$$

denn sie erfüllt die Behauptung des Pumping-Lemmas: Dies ist klar für Wörter aus $\{a, b\}^*$. Sei $n_0 \geq 1$ die Pumping Lemma Zahl zu $L$ und sei nun $x = c^m a^n b^n$ ein Wort aus

$L$ der Länge $\geq n_0$. Dann ist zum Beispiel $u = \varepsilon$, $v = c$, $w = c^{m-1}a^n b^n$ eine Zerlegung, die die Eigenschaften 1,2,3 besitzt. (Man beachte, dass dies auch für den Fall $m = 1$ gilt).

*Graphische Darstellung:*




*Beispiel 1:* Die Sprache

$$L = \{a^n b^n \mid n \geq 1\}$$

ist *nicht* regulär (aber kontextfrei). Dies beweist man mit dem Pumping Lemma so: Angenommen $L$ sei regulär. Dann gibt es eine Zahl $n_0$, so dass sich alle Wörter $x \in L$ der Länge $\geq n_0$ wie im Pumping Lemma beschrieben zerlegen lassen. Betrachten wir speziell das Wort $a^{n_0}b^{n_0}$ der Länge $2n_0 \geq n_0$. Die entsprechende Zerlegung $uvw$ dieses Wortes ist aufgrund Bedingung 1 so, dass $v$ nicht leer ist, und aufgrund Bedingung 2 kann $v$ nur aus $a$'s bestehen. Aufgrund von Bedingung 3 wäre dann auch das Wort $uw = a^{n_0-|v|}b^n$ in der Sprache, was im Widerspruch zur Definition von $L$ steht. Daher ist $L$ nicht regulär.

*Beispiel 2:* Die Sprache

$$L = \{0^m \mid m \text{ ist Quadratzahl}\}$$

ist nicht regulär: Angenommen doch, dann gibt es gemäß Pumping Lemma eine entsprechende Zahl $n_0$, so dass sich jedes Wort der Form $0^m, m \geq n_0$, $m$ Quadratzahl, zerlegen lässt in $uvw$ mit den Eigenschaften 1,2,3. Wähle speziell $x = 0^{n_0^2}$. Betrachte eine beliebige Zerlegung $x = uvw$, die die Bedingungen 1,2,3 erfüllt. Wegen Bedingung 1 und 2 gilt:

$$1 \leq |v| \leq |uv| \leq n_0$$

Mit Bedingung 3, $i = 2$, gilt dann:

$$uv^2w \in L$$

Andererseits gilt die Abschätzung:

$$n_0^2 = |x| = |uvw| < |uv^2w| \leq n_0^2 + n_0 < n_0^2 + 2n_0 + 1 = (n_0 + 1)^2$$

Somit kann die Zahl $|uv^2w|$ keine Quadratzahl sein, da sie echt zwischen zwei aufeinanderfolgenden Quadratzahlen liegt. Widerspruch! Also ist gezeigt, dass $L$ nicht regulär ist.

*Beispiel 3:* Die Sprache

$$L = \{0^p \mid p \text{ ist Primzahl}\}$$

ist nicht regulär. Sei wieder angenommen, $L$ ist regulär und sei $n$ die entsprechende Pumping-Lemma-Zahl. Da es unendlich viele Primzahlen gibt, gibt es auch solche $\geq n+2$. Sei also $p \geq n+2$ eine Primzahl, also $0^p \in L$. $0^p$ müsste sich nun in $uvw$ zerlegen lassen mit den Bedingungen 1, 2 und 3. Aufgrund Bedingung 3 müssten alle Wörter der Form $uv^iw = 0^{|uw|+i\cdot|v|}$ in $L$ liegen. Das heißt, alle Zahlen der Form $|uw| + i \cdot |v|$ müssten Primzahlen sein. (Man beachte, dass wegen der Bedingung 1 $|v| \geq 1$ gilt). Wähle nun speziell $i = |uw|$. Dann erhalten wir: $|uw| + |uw| \cdot |v|$ = $|uw| \cdot (1 + |v|)$. Diese Zahl lässt sich also in zwei nicht-triviale Faktoren zerlegen, kann also keine Primzahl sein. (Man beachte, dass wegen der Länge von $x$ und wegen Bedingung 2 gilt $|uw| \geq |w| = |x| - |uv| \geq 2$.) Widerspruch! Also kann $L$ nicht regulär sein.

### 3.2.5 Äquivalenzrelationen und Minimalautomaten

Man kann jeder Sprache $L$ eine Äquivalenzrelation $R_L$ auf $\Sigma^*$ zuordnen. (Zu Äquivalenzrelationen siehe den mathematischen Anhang).

**Definition.** Es gilt $xR_Ly$ genau dann, wenn für alle Wörter $z \in \Sigma^*$ gilt:

$$xz \in L \Leftrightarrow yz \in L$$

Man verifiziert sofort, dass $R_L$ die Axiome einer Äquivalenzrelation erfüllt. Intuitiv sind zwei Wörter $x$ und $y$ dann äquivalent, wenn sich bei Anfügen von beliebigen $z$ die Wörter $xz$ und $yz$ bzgl. Mitgliedschaft in $L$ gleich verhalten. Da der Fall $z = \varepsilon$ mit eingeschlossen ist, muss also insbesondere auch $x \in L \Leftrightarrow y \in L$ gelten. Entscheidend ist für das Folgende, ob der Index (die Anzahl der erzeugten Äquivalenzklassen) von $R_L$ endlich oder unendlich ist.

**Satz (Myhill, Nerode).**
EINE SPRACHE $L$ IST GENAU DANN REGULÄR, WENN DER INDEX VON $R_L$ ENDLICH IST.

**Beweis:** (⇒) Sei $L$ regulär und sei $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA mit $L = T(M)$. Dann kann man $M$ eine Äquivalenzrelation $R_M$ wie folgt zuordnen: Es gilt $xR_My$, falls $\hat{\delta}(z_0, x) = \hat{\delta}(z_0, y)$, d.h. falls die Eingaben $x$ und $y$ den Automaten in denselben Zustand

überführen. Wir zeigen $R_M \subseteq R_L$, d.h. $R_M$ ist eine *Verfeinerung* von $R_L$. Es gelte $xR_My$, also $\hat{\delta}(z_0, x) = \hat{\delta}(z_0, y)$. Sei nun $z \in \Sigma^*$ beliebig. Dann gilt

$$\begin{aligned} xz \in L &\Leftrightarrow \hat{\delta}(z_0, xz) \in E \\ &\Leftrightarrow \hat{\delta}(\hat{\delta}(z_0, x), z) \in E \\ &\Leftrightarrow \hat{\delta}(\hat{\delta}(z_0, y), z) \in E \\ &\Leftrightarrow \hat{\delta}(z_0, yz) \in E \\ &\Leftrightarrow yz \in L \end{aligned}$$

Somit gilt:

$$\begin{aligned} Index(R_L) &\leq Index(R_M) \\ &= \text{Anzahl der Zustände, die von } z_0 \text{ aus erreichbar sind} \\ &\leq |Z| \\ &< \infty \end{aligned}$$

($\Leftarrow$) Wenn der Index von $R_L$ endlich ist, so gibt es endlich viele Wörter $x_1, x_2, \ldots, x_k$ mit $\Sigma^* = [x_1] \cup [x_2] \cup \cdots \cup [x_k]$. Definiere nun einen DFA, dessen Zustände gerade durch diese Äquivalenzklassen identifiziert werden:

$$M = (Z, \Sigma, \delta, z_0, E)$$

wobei

$$\begin{aligned} Z &= \{[x_1], \ldots, [x_k]\} \\ \delta([x], a) &= [xa] \\ z_0 &= [\varepsilon] \\ E &= \{[x] \mid x \in L\} \end{aligned}$$

Aus der Definition von $\delta$ erhalten wir: $\hat{\delta}([\varepsilon], x) = [x]$. Zusammen mit der Definition von $z_0$ und $E$ ergibt sich dann

$$\begin{aligned} x \in T(M) &\Leftrightarrow \hat{\delta}(z_0, x) \in E \\ &\Leftrightarrow \hat{\delta}([\varepsilon], x) \in E \\ &\Leftrightarrow [x] \in E \\ &\Leftrightarrow x \in L. \end{aligned}$$

□

*Beispiel:* Betrachten wir noch einmal die nicht-reguläre Sprache

$$L = \{a^n b^n \mid n \geq 1\}$$

und sehen wir uns einige der Äquivalenzklassen von $R_L$ an:

$$\begin{aligned}
[ab] &= L \\
[a^2b] &= \{a^2b, a^3b^2, a^4b^3, \ldots\} \\
[a^3b] &= \{a^3b, a^4b^2, a^5b^3, \ldots\} \\
&\vdots \\
[a^kb] &= \{a^{k+i-1}b^i \mid i \geq 1\} \\
&\vdots
\end{aligned}$$

Man erkennt, dass für $i \neq j$ die Wörter $a^ib$ und $a^jb$ nicht äquivalent sind, denn mit $z = b^{i-1}$ gilt: $a^ibz \in L$ und $a^jbz \notin L$. Somit sind $[ab], [a^2b], [a^3b], \ldots$ paarweise verschiedene (damit disjunkte) Äquivalenzklassen. Somit ist $Index(R_L) = \infty$ und $L$ ist nicht regulär.

Man beachte, dass es für einen Beweis der Nicht-Regularität nicht unbedingt notwendig ist, die Äquivalenzklassenstruktur von $R_L$ vollständig aufzuklären. Es genügt, unendlich viele nicht-äquivalente Wörter aus $\Sigma^*$ zu identifizieren, um auf $Index(R_L) = \infty$ zu schließen.

*Beispiel:* Betrachte die Sprache

$$L = \{x \in \{0,1\}^* \mid x \text{ endet mit } 00\}$$

Es gilt:

$$\begin{aligned}
[\varepsilon] &= \{x \mid x \text{ endet nicht mit } 0\} \\
[0] &= \{x \mid x \text{ endet mit } 0, \text{ aber nicht mit } 00\} \\
[00] &= \{x \mid x \text{ endet mit } 00\}
\end{aligned}$$

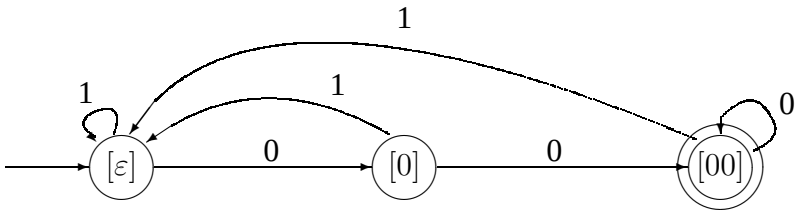
Dies sind alle Äquivalenzklassen von $R_L$, also

$$\Sigma^* = [\varepsilon] \cup [0] \cup [00]$$

Der „Äquivalenzklassenautomat“, der im obigen Beweis konstruiert wird, hat die drei Zustände $[\varepsilon], [0], [00]$. Es gilt:

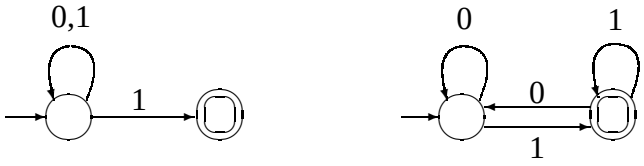
$$\begin{aligned}
\delta([\varepsilon], 0) &= [0] \\
\delta([\varepsilon], 1) &= [1] = [\varepsilon] \\
\delta([0], 0) &= [00] \\
\delta([0], 1) &= [01] = [\varepsilon] \\
\delta([00], 0) &= [000] = [00] \\
\delta([00], 1) &= [001] = [\varepsilon] \\
z_0 &= [\varepsilon] \\
E &= \{[00]\}
\end{aligned}$$

Bildlich dargestellt ist dies der folgende Automat:




*Bemerkung:* Der Äquivalenzklassenautomat ist der Automat mit der kleinsten Anzahl von Zuständen, der sog. *Minimalautomat*, denn aus obigem Beweis ergibt sich, dass für jeden beliebigen Automaten $M$ für die gegebene Sprache $L$ gilt: $R_M \subseteq R_L = R_{M_0}$, wobei $M_0$ der Äquivalenzklassenautomat ist. Das heißt, die dem Automaten $M$ zugeordnete Äquivalenzrelation $R_M$ ist eine Verfeinerung der Äquivalenzrelation $R_L = R_{M_0}$. Das bedeutet, dass die Zahl der Zustände von $M$ größer oder gleich der von $M_0$ ist. Außerdem erkennt man, dass es keine zwei strukturell unterschiedlichen Automaten für $L$ mit minimaler Zustandszahl geben kann. Denn wenn $M$ die minimale Zustandszahl besitzt, so müssen $R_M$ und $R_L$ identisch sein. Der Minimalautomat ist also bis auf Isomorphie (d.h. Umbenennen der Zustände) eindeutig bestimmt.

Man beachte, dass diese letzte Aussage für NFA's nicht richtig ist. Es gibt strukturell unterschiedliche NFA's mit minimaler Zustandszahl. Die folgenden beiden nichtisomorphen NFA's mit 2 Zuständen erkennen beide die Menge aller Wörter, die mit 1 enden (wobei der rechte Automat der minimale DFA ist).




Wie kann man von einem gegebenen DFA feststellen, ob er bereits minimal ist? Aufgrund des obigen Beweises ist ein Automat $M$ mit $T(M) = L$ offensichtlich dann nicht minimal, wenn es zwei verschiedene Zustände $z, z'$ gibt, so dass für alle $x$ gilt:

$$\hat{\delta}(z, x) \in E \Leftrightarrow \hat{\delta}(z', x) \in E.$$

In diesem Fall können die Zustände $z$ und $z'$ zu einem einzigen Zustand „verschmolzen" werden. Ferner überlegt man sich, dass es bei diesen Tests genügt, Wörter $x$ zu betrachten, deren Länge durch die Anzahl der Zustände von $M$ beschränkt ist.

Der folgende Algorithmus macht von diesen Überlegungen in effizienter Weise Gebrauch. Wir verzichten auf seine Analyse, die sich aber im wesentlichen leicht aus dem obigen Beweis ergibt.

**Algorithmus Minimalautomat**

*Eingabe:* ein DFA $M$ (Zustände, die vom Startzustand aus nicht erreichbar sind, sind bereits entfernt).
*Ausgabe:* Angabe, welche Zustände von $M$ noch zu verschmelzen sind, um den Minimalautomaten zu erhalten.

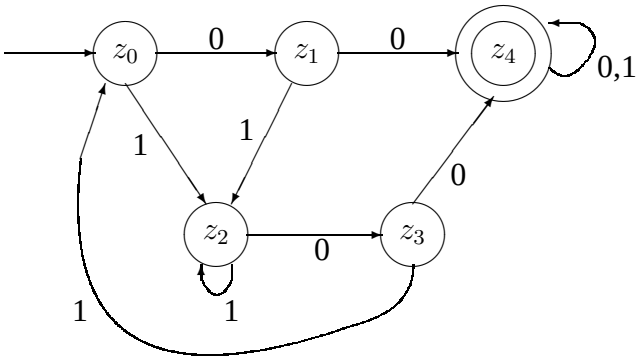
1. Stelle eine Tabelle aller Zustandspaare $\{z, z'\}$ mit $z \neq z'$ von $M$ auf.
2. Markiere alle Paare $\{z, z'\}$ mit $z \in E$ und $z' \notin E$ (oder umgekehrt).
3. Für jedes noch unmarkierte Paar $\{z, z'\}$ und jedes $a \in \Sigma$ teste, ob

   $$\{\delta(z, a), \delta(z', a)\}$$

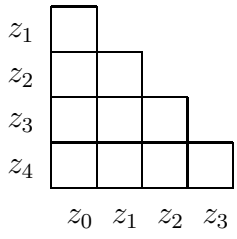
   bereits markiert ist. Wenn ja: markiere auch $\{z, z'\}$.
4. Wiederhole den letzten Schritt, bis sich keine Änderung in der Tabelle mehr ergibt.
5. Alle jetzt noch unmarkierten Paare können jeweils zu einem Zustand verschmolzen werden.

*Bemerkung:* Der oben beschriebene Algorithmus hat – geeignet implementiert (siehe Hopcroft/Ullman) – die Zeitkomplexität $O(n^2)$.

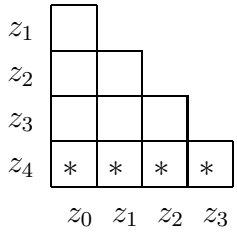
*Beispiel:* Gegeben sei folgender DFA:



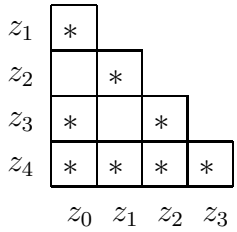

Wir führen nun die Schritte durch: Zunächst eine Tabelle der Zustandspaare anfertigen:



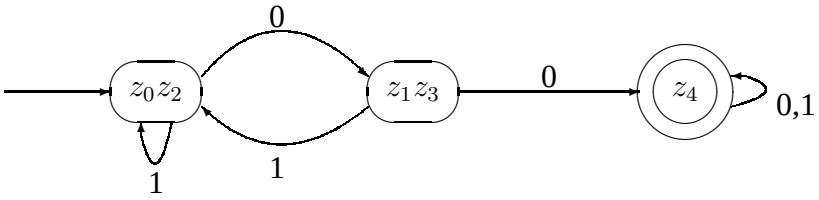

Nächster Schritt: Paare von Endzuständen und Nicht-Endzuständen markieren.




Gemäß Regel 3 weiter markieren ...




Es können also die Zustände $z_0, z_2$ und $z_1, z_3$ in jeweils einen Zustand verschmolzen werden. Der resultierende Minimalautomat:




Dieser Automat beschreibt offensichtlich die Sprache

$$L = \{x \in \{0,1\}^* \mid \text{ in } x \text{ kommt 00 vor}\}$$

Die drei Zustände $z_0z_2$, $z_1z_3$ und $z_4$ entsprechen den Äquivalenzklassen $[\varepsilon]$, $[0]$ und $[00]$.

### 3.2.6 Abschlusseigenschaften

Wir wollen nun untersuchen, unter welchen Operationen die regulären Sprachen abgeschlossen sind (vgl. hierzu den mathematischen Anhang). Einige der folgenden Abschlusseigenschaften sind eine triviale Konsequenz dessen, dass die regulären Ausdrücke gerade die regulären Sprachen beschreiben.

**Satz.**
DIE REGULÄREN SPRACHEN SIND ABGESCHLOSSEN UNTER:

- VEREINIGUNG
- SCHNITT
- KOMPLEMENT
- PRODUKT
- STERN

**Beweis:** Der Abschluss unter Vereinigung, Produkt und Stern-Operation folgt unmittelbar daraus, dass mit zwei durch die regulären Ausdrücke $\alpha$ und $\beta$ gegebenen regulären Sprachen auch die durch $(\alpha|\beta)$, $\alpha\beta$ und $(\alpha)^*$ bezeichneten Sprachen regulär sind (vgl. Satz von Kleene, S. 75).

Als nächstes betrachten wir den Abschluss unter Komplementbildung. Sei eine reguläre Sprache $A$ durch ihren DFA $M = (Z, \Sigma, \delta, z_0, E)$ gegeben. Indem wir Endzustände mit Nicht-Endzuständen vertauschen, erhalten wir $M' = (Z, \Sigma, \delta, z_0, Z - E)$. Es ist klar, dass $T(M') = \overline{A}$, und damit die regulären Sprachen unter Komplement abgeschlossen sind.

Mit dem Abschluss unter Vereinigung und unter Komplement sind die regulären Sprachen vermittels der deMorganschen Regeln auch „automatisch“ unter Schnitt abgeschlossen (vgl. den mathematischen Anhang). Wir geben jedoch trotzdem eine direkte – weil interessante – Konstruktion an: Seien $M_1 = (Z_1, \Sigma, \delta_1, z_{01}, E_1)$ und $M_2 = (Z_2, \Sigma, \delta_2, z_{02}, E_2)$ DFAs. Wir konstruieren den „Kreuzproduktautomaten“ $M$ aus $M_1$ und $M_2$. Dieser akzeptiert gerade die Sprache $T(M_1) \cap T(M_2)$.

$$M = (Z_1 \times Z_2, \Sigma, \delta, (z_{01}, z_{02}), E_1 \times E_2)$$

wobei

$$\delta((z, z'), a) = (\delta_1(z, a), \delta_2(z', a))$$

□

*Bemerkung:* Da die regulären Ausdrücke gerade die regulären Sprachen beschreiben, muss es zu jedem regulären Ausdruck $\alpha$ einen Ausdruck $\beta$ geben, der die Komplementsprache darstellt: $L(\beta) = \overline{L(\alpha)}$. Das heißt, die Komplementbildung ist mittels Vereinigung, Produkt und Sternoperation darstellbar. Die entsprechende Konstruktion (so wie sie hier angegeben ist) ist jedoch sehr aufwändig, da sie die Stufen *regulärer Ausdruck* → *NFA* → *DFA* → *komplementärer DFA* → *regulärer Ausdruck* durchlaufen müsste.

### 3.2.7 Entscheidbarkeit

Wir wollen einige Fragen bzgl. Entscheidbarkeit bei regulären Sprachen diskutieren.

Das *Wortproblem* (gegeben: $x$ ; gefragt: liegt $x$ in $L(G)$ bzw. $T(M)$) ist, wie bereits gezeigt, für Typ 1,2,3 Grammatiken entscheidbar. Sollte die zu entscheidende reguläre Sprache durch einen DFA gegeben sein, so ist das Wortproblem sogar in linearer Zeit lösbar: Verfolge Zeichen für Zeichen die Zustandsübergänge im Automaten, die durch die Eingabe $x$ hervorgerufen werden. Falls ein Endzustand erreicht wird, liegt $x$ in der Sprache.

Das *Leerheitsproblem* stellt bei gegebenem $G$ (bzw. $M$) die Frage, ob $L(G) = \emptyset$ (bzw. $T(M) = \emptyset$). Auch dies ist entscheidbar: Sei $M$ ein gegebener DFA (oder NFA) für die Sprache. $T(M)$ ist offensichtlich genau dann leer, wenn es keinen Pfad vom Startzustand (von einem Startzustand) zu einem Endzustand gibt.

Das *Endlichkeitsproblem* stellt bei gegebenem $G$ (bzw. $M$) die Frage, ob $|L(G)| < \infty$ (bzw. $|T(M)| < \infty$), also ob die definierte Sprache endlich oder unendlich ist.

Sei $n$ die Pumping Lemma Zahl zu $L$. Es gilt:

$$|L| = \infty \quad \Leftrightarrow \quad \text{es gibt ein Wort der Länge} \geq n \text{ und} < 2n \text{ in } L$$

Begründung: ($\Leftarrow$) Sei $x \in L$ mit $n \leq |x| < 2n$. Aufgrund des Pumping Lemmas lässt sich $x$ zerlegen in $uvw$, so dass $\{uv^i w \mid i \geq 0\}$ Teilmenge von $L$ ist. Also ist $|L|$ unendlich.
($\Rightarrow$) Sei $|L| = \infty$ und sei angenommen, das kürzeste Wort $x \in L$ mit Länge $\geq n$ habe Länge $\geq 2n$. Aufgrund des Pumping Lemmas lässt sich $x$ zerlegen in $uvw$, so dass auch $uv^0w = uw$ Element von $L$ ist. Da $|v| \leq |uv| \leq n$, hat dieses Wort eine Länge $\geq n$. Dies widerspricht der Minimalität von $x$. Daher gibt es ein Wort mit Länge zwischen $n$ und $2n$.

Der Entscheidungsalgorithmus für $|L| \stackrel{?}{<} \infty$ verläuft also so, dass alle Wörter $x$ der Länge $\geq n$ und $< 2n$ auf Mitgliedschaft in $L$ ($\rightarrow$Wortproblem) geprüft werden müssen.

Dieses Argument zeigt also, dass das Endlichkeitsproblem entscheidbar ist, indem das Problem vermittels des Pumping Lemmas auf das Wortproblem zurückgeführt wird. Unter Effizienzaspekten (um die geht es hier aber eigentlich nicht) ist der betreffende Algorithmus jedoch hoffnungslos ineffizient, da exponentiell viele Wörter getestet werden müssen. Es geht aber auch einfacher: Es ist $|T(M)| = \infty$ genau dann, wenn es in dem vom Startzustand erreichbaren Teilgraphen einen Zyklus gibt. Dies kann durch eine einfache „depth-first“ Suche effizient festgestellt werden.

Das *Schnittproblem* stellt bei gegebenen $G_1, G_2$ (bzw. $M_1, M_2$) die Frage ob $L(G_1) \cap L(G_2)$ (bzw. $T(M_1) \cap T(M_2)$) leer ist oder nicht.

Da die regulären Sprachen *effektiv* unter Schnitt abgeschlossen sind (d.h. zu gegebenen regulären Grammatiken $G_1, G_2$ ist algorithmisch eine reguläre Grammatik $G$ erzeugbar mit $L(G) = L(G_1) \cap L(G_2)$), lässt sich die Frage auf das Leerheitsproblem reduzieren und ist damit entscheidbar.

Beim *Äquivalenzproblem* sind zwei reguläre Grammatiken bzw. endliche Automaten gegeben und es ist gefragt, ob diese dieselbe Sprache definieren.

Falls zwei DFAs vorliegen, lässt sich die Frage dadurch beantworten, dass zu jedem der

Automaten der Minimalautomat konstruiert wird und diese dann auf Isomorphie verglichen werden.

Eine andere Lösung geht von der Tatsache aus, dass die regulären Sprachen effektiv unter Schnitt, Vereinigung und Komplementbildung abgeschlossen sind. Es gilt

$$L_1 = L_2 \;\Leftrightarrow\; (L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1}) = \emptyset$$

Daher lässt sich das Problem auf die Entscheidbarkeit des Leerheitsproblems zurückführen.

## 3.3 Kontextfreie Sprachen

Die regulären Sprachen und die zugehörigen Formalismen, wie endliche Automaten, sind – wie sich gezeigt hat – sehr nützlich. Jedoch sind die Grenzen dieses einfachen Formalismus auch schnell erreicht. Die nächstgrößere Klasse der Chomsky-Hierarchie bilden die kontextfreien Sprachen. Diese sind besonders nützlich, um geklammerte Sprachstrukturen, wie sie insbesondere bei Programmiersprachen auftreten, zu beschreiben.

Typische Beispiele hierzu sind die Grammatiken für die (beliebig geklammerten) arithmetischen Ausdrücke

$$E \to T \mid E + T,\; T \to F \mid T * F,\; F \to a \mid (E)$$

sowie für die Anweisungen (etwa in Modula):

$$\begin{aligned} < \text{Anw} > &\to < \text{While-Anw} > \mid < \text{If-Anw} > \\ < \text{While-Anw} > &\to \text{WHILE } < \text{Bedingung} > \text{ DO } < \text{Anw} > \text{ END} \\ < \text{If-Anw} > &\to \text{IF } < \text{Bedingung} > \text{ THEN } < \text{Anw} > \text{ END} \\ < \text{Bedingung} > &\to \ldots \end{aligned}$$

Die öffnenden und schließenden Klammerpaare, die hier erkennbar sind, sind ( und ), DO und END, sowie THEN und END.

Die nicht-reguläre Beispielsprache

$$L = \{a^n b^n \mid n \geq 1\}$$

ist, wie man leicht sieht, kontextfrei:

$$S \to ab|aSb$$

Dies beweist, wie bereits angekündigt, dass die Typ 3 Sprachen echt in der Klasse der Typ 2 Sprachen enthalten sind.

### 3.3.1 Normalformen

Wir untersuchen nun, auf welche möglichst einfachen Formen bei den Regeln von kontextfreien Grammatiken man sich beschränken kann. Das heißt, es geht darum, nachzuweisen, dass jede kontextfreie Grammatik in eine äquivalente solche umgeformt werden kann, so dass alle Regeln eine sog. Normalform besitzen.

Wir erinnern zunächst daran, dass jede kontextfreie Grammatik „$\varepsilon$-frei“ gemacht werden kann (vgl. S. 59). Wir betrachten nun eine weitere allgemeine Umformung, die zum Ziel hat, alle Regeln der Form $A \to B$ zu eliminieren, wobei $A, B$ Variablen sind. Zunächst ist algorithmisch einfach feststellbar, ob es eine Menge von Variablen $B_1, \ldots, B_k$ gibt mit $B_1 \to B_2$, …, $B_{k-1} \to B_k$ und $B_k \to B_1$. In diesem Fall ersetzen wir die Variablen $B_1, \ldots, B_k$ durch eine einzige Variable $B$.

Als nächstes können die Variablen so durchnummeriert werden, $V = \{A_1, A_2, \ldots, A_n\}$, dass aus $A_i \to A_j \ \in P$ folgt $i < j$. Wir gehen nun *von hinten nach vorne* vor und eliminieren für $k = n-1, \ldots, 1$ alle Regeln der Form $A_k \to A_{k'},\ k' > k$. Seien die Regeln mit $A_{k'}$ auf der linken Seite gegeben durch

$$A_{k'} \to x_1 | x_2 | \ldots | x_m.$$

Wir fügen dann die Regeln

$$A_k \to x_1 | x_2 | \ldots | x_m$$

hinzu. Diese Umformung leistet offenbar das Gewünschte.

Nach diesen Vorbemerkungen definieren wir nun die wichtigste Normalform, die *Chomsky Normalform* (kurz: CNF).

> **Definition.** Eine kontextfreie Grammatik $G$ mit $\varepsilon \notin L(G)$ heißt in *Chomsky Normalform*, falls alle Regeln eine der beiden Formen haben:
>
> $$A \to BC$$
>
> bzw.
>
> $$A \to a$$
>
> Hierbei stehen $A, B, C$ für Variablen und $a$ für ein Terminalsymbol.

Offensichtlich bedeutet dies eine starke Einschränkung für eine kontextfreie Grammatik: Ableitungsbäume für solche Grammatiken sind – bis auf den letzten Ableitungsschritt – *Binärbäume*. Man überlegt sich ferner, dass ein Wort $w \in L(G)$ in genau $2|w| - 1$ Ableitungsschritten erzeugt wird.

**Satz.**
ZU JEDER KONTEXTFREIEN GRAMMATIK $G$ MIT $\varepsilon \notin L(G)$ GIBT ES EINE CHOMSKY NORMALFORM GRAMMATIK $G'$ MIT $L(G) = L(G')$.

**Beweis:** Nach der oben diskutierten Vorverarbeitung hat jede Regel der gegebenen kontextfreien Grammatik $G = (V, \Sigma, P, S)$ eine der Formen:

$$A \to a \quad (A \in V,\ a \in \Sigma)$$

bzw.

$$A \to x \quad (A \in V,\ x \in (V \cup \Sigma)^*,\ |x| \geq 2)$$

Für jedes Terminalzeichen $a \in \Sigma$ fügen wir eine neue Variable $B$ zu $V$ hinzu, sowie zu $P$ die Regel

$$B \to a$$

hinzu.

Als nächstes ersetzen wir jedes Terminalzeichen $a$ auf der rechten Seite einer Regel durch die zugehörige, gerade eingeführte Variable $B$ (außer die Regel hat bereits die Form $A \to a$).

Nun haben alle Regeln die Form

$$A \to a$$

oder

$$A \to B_1 B_2 \ldots B_k\ ,\ k \geq 2.$$

Nicht in Chomsky Normalform sind jetzt nur noch Regeln der Form

$$A \to B_1 B_2 \ldots B_k\ ,\ k \geq 3.$$

Für jede solche Regel führen wir weitere neue Variablen $C_2, \ldots, C_{k-1}$ ein und ersetzen solche Regeln durch

$$\begin{aligned} A &\to B_1 C_2 \\ C_2 &\to B_2 C_3 \\ &\vdots \\ C_{k-1} &\to B_{k-1} B_k \end{aligned}$$

□

Ein Beispiel für die Umformung in CNF findet sich – im Zusammenhang mit dem CYK-Algorithmus – auf Seite 98.

Wir geben noch eine weitere Normalform für kontextfreie Grammatiken an, die sog. *Greibach Normalform* (kurz: GNF).

> **Definition.** Eine kontextfreie Grammatik $G$ mit $\varepsilon \notin L(G)$ heißt in *Greibach Normalform*, falls alle Regeln die Form haben:
>
> $$A \to a B_1 B_2 \ldots B_k \quad (k \geq 0)$$
>
> Hierbei stehen $A, B_1, \ldots, B_k$ für Variablen und $a$ für ein Terminalsymbol.

Die Greibach Normalform ist eine natürliche Erweiterung der regulären Grammatik, dort wäre nur der Fall $k = 0$ und $k = 1$ zugelassen.

**Satz.**
ZU JEDER KONTEXTFREIEN GRAMMATIK $G$ MIT $\varepsilon \notin L(G)$ GIBT ES EINE GREIBACH NORMALFORM GRAMMATIK $G'$ MIT $L(G) = L(G')$.

(ohne Beweis)

Wir bemerken abschließend noch, dass man jede kontextfreie Grammatik so in Greibach Normalform umformen kann, dass auf der rechten Seite der Regeln nicht mehr als 2 Variablen vorkommen.

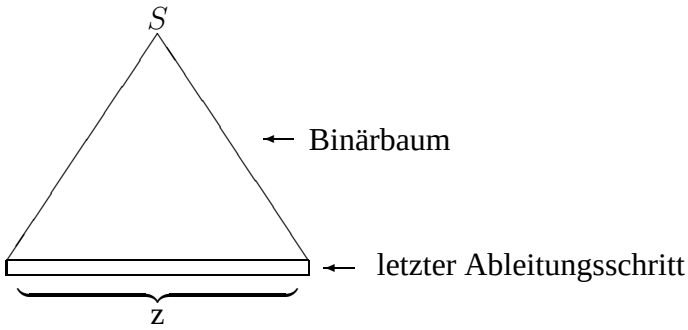
### 3.3.2 Das Pumping Lemma

Wir beweisen nun in völliger Analogie zum Pumping Lemma für reguläre Sprachen eine entsprechende Version für kontextfreie Sprachen. Das Pumping Lemma stellt das Haupthilfsmittel dar, um von einer Sprache nachzuweisen, dass sie *nicht* kontextfrei ist.

**Satz (Pumping Lemma, $uvwxy$-Theorem).**
SEI $L$ EINE KONTEXTFREIE SPRACHE. DANN GIBT ES EINE ZAHL $n \in \mathbb{N}$, SO DASS SICH ALLE WÖRTER $z \in L$ MIT $|z| \geq n$ ZERLEGEN LASSEN IN $z = uvwxy$ MIT FOLGENDEN EIGENSCHAFTEN:

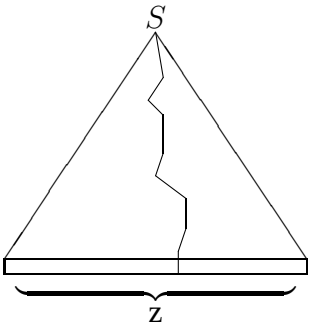
1. $|vx| \geq 1$
2. $|vwx| \leq n$
3. FÜR ALLE $i \geq 0$ GILT: $uv^iwx^iy \in L$

**Beweis:** Sei $G$ eine Grammatik für $L - \{\varepsilon\}$ in Chomsky Normalform. Sei $k$ die Anzahl der Variablen in $G$. Wähle $n = 2^k$. Betrachte nun den Ableitungsbaum eines beliebigen Wortes $z \in L(G)$ mit $|z| \geq n$. Dieser ist – bis auf den letzten Ableitungsschritt – ein Binärbaum:

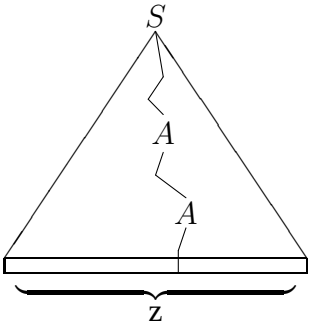


Dieser Binärbaum hat $\geq 2^k$ Blätter. Daher muss mindestens ein Pfad der Länge $\geq k$ existieren (siehe separates, nächstes Lemma).

Halte einen solchen Pfad maximaler Länge fest:

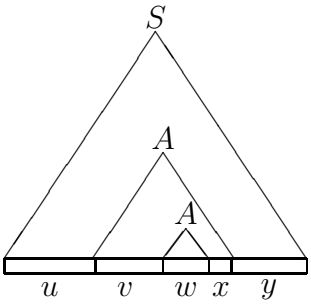



Einschließlich der Startvariablen befinden sich auf diesem Pfad $\geq k+1$ Variablen. Da die Grammatik nur $k$ Variablen besitzt, muss mindestens eine Variable doppelt vorkommen (Schubfachschluss):




Um ein solches Doppelvorkommen einer Variablen $A$ zu bestimmen, gehen wir hier immer *von unten nach oben* vor, bis eine Variable zum zweiten Mal auf dem Pfad erscheint. Dieses Vorgehen garantiert, dass das obere der beiden $A$'s höchstens $k$ Schritte von der Blattebene entfernt ist.

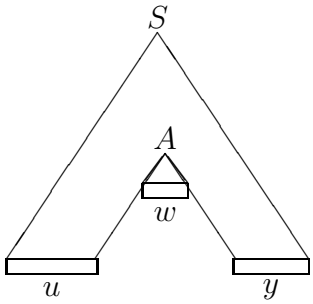
Betrachten wir nun die Teilwörter, die aus den beiden $A$'s abgeleitet werden können. Diese induzieren eine Zerlegung von $z$ in Teilwörter $uvwxy$ wie folgt:



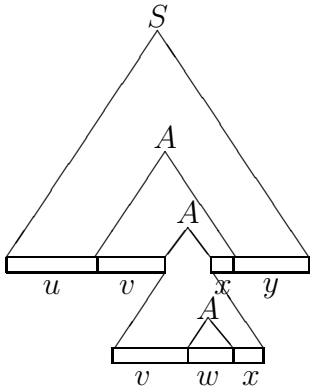
Aus der Tatsache, dass das obere $A$ wegen der Chomsky Normalform zunächst mittels einer Regel der Form $A \to BC$ weiter abgeleitet wird, folgt dass $v$ oder $x$ (oder beide) nicht leer ist. Entweder geht $v$ aus $B$ hervor oder $x$ aus $C$. Damit ist Eigenschaft 1 der $uvwxy$-Zerlegung von $z$ bestätigt.

Die Eigenschaft 2 ergibt sich daraus, dass der Abstand des oberen $A$'s von der Blattebene höchstens $k$ ist (siehe oben). Deshalb kann das vom oberen $A$ abgeleitete Wort $vwx$ höchstens die Länge $2^k = n$ haben (vgl. nachfolgendes Lemma).

Die Ableitungsfolge (bzw. der Ableitungsbaum) für $z$ kann aufgrund des Doppelvorkommens der Variablen $A$ auf verschiedene Arten modifiziert werden. Zum Beispiel kann an das obere $A$ der Ableitungsbaum des unteren $A$ gehängt werden. Wir erhalten damit eine Ableitung von $uwy = uv^0wx^0y$. Das heißt, $uv^0wx^0y \in L(G)$.




Man kann auch an das untere $A$ den Teilbaum, der am oberen $A$ beginnt, hängen:



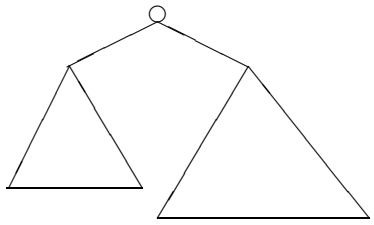

Dies ergibt eine Ableitung von $uvvwxxy = uv^2wx^2y$. Allgemein gilt, dass für jedes $i \geq 0$ $uv^iwx^iy$ in $L(G)$ liegt. Damit ist die Eigenschaft 3 bewiesen. □

Das folgende Lemma wird im Beweis des Pumping Lemmas benötigt:

> **Lemma.** Sei $B$ ein Binärbaum (gemeint ist: jeder Knoten in $B$ hat entweder 0 oder 2 Söhne) mit $\geq 2^k$ Blättern.
>
> Dann hat $B$ mindestens einen Pfad der Länge $\geq k$.

**Beweis:** Induktion über $k$: Jeder Binärbaum mit mindestens $2^0 = 1$ Blättern hat trivialerweise mindestens einen Pfad der Länge 0.

Gelte nun die Behauptung für ein (beliebiges, aber festes) $k$. Betrachte nun einen Binärbaum mit $\geq 2^{k+1}$ Blättern. Mindestens einer seiner zwei Teilbäume hat $\geq 2^{k+1}/2 = 2^k$ Blätter.



$\geq 2^k$ Blätter

Somit existiert in diesem Unterbaum ein Pfad der Länge $\geq k$. Dieser Pfad kann zur Wurzel des Gesamtbaumes um 1 verlängert werden und liefert somit einen Pfad der gewünschten Länge $\geq k + 1$. □

Es gelten sinngemäß dieselben Bemerkungen über die einseitige Implikation des Pumping Lemmas wie bei den regulären Sprachen:

$$(L \text{ kontextfrei}) \implies (\exists n \in I\!N)(\forall z \in L,\ |z| \geq n)(\exists u, v, w, x, y) \\ [(z = uvwxy) \wedge 1 \wedge 2 \wedge 3]$$

*Beispiel:* Die Sprache

$$L = \{a^m b^m c^m \mid m \geq 1\}$$

ist nicht kontextfrei (aber kontextsensitiv).

Angenommen $L$ sei kontextfrei. Dann liefert uns das Pumping Lemma eine Zahl $n$, so dass sich alle Wörter $z = a^m b^m c^m$ mit $|z| \geq n$ zerlegen lassen in $uvwxy$ mit den Eigenschaften 1,2,3. Wähle etwa $z = a^n b^n c^n$. Dann ist $|z| = 3n \geq n$. Wegen Eigenschaft 2 kann $vx$ nicht aus $a$'s, $b$'s *und* $c$'s bestehen. (Wegen der Längenbeschränkung von $vwx$ kann sich dieses Wort nicht über den gesamten Bereich der $b$'s hinweg erstrecken).

Wegen Eigenschaft 1 ist $vx$ nicht leer, und wegen Eigenschaft 3 (mit $i = 0$) muss $uv^0wx^0y = uwy$ in $L$ liegen. Wegen der obigen Diskussion kann $uwy$ also nicht die Form $a^m b^m c^m$ haben. Dieser Widerspruch beweist, dass $L$ nicht kontextfrei ist.

Dass $L$ kontextsensitiv ist, wurde schon zuvor gezeigt (Seite 57). Damit sind die kontextfreien Sprachen tatsächlich *echt* in den kontextsensitiven Sprachen enthalten.

Weitere *Beispiele:* Die Sprachen

$$\{0^p \mid p \text{ ist Primzahl}\}$$

und

$$\{0^n \mid n \text{ ist Quadratzahl}\}$$

sind nicht kontextfrei. Da hier nur Wörter über dem einelementigen Alphabet $\{0\}$ vorkommen, spielt es in der Aussage des Pumping Lemmas keine Rolle, an welcher Position im Wort $z$ die Teilwörter $v$ und $x$ vorkommen. Daher können diese zusammengefasst werden und in diesem Fall sind die Behauptungen des Pumping Lemmas für kontextfreie Sprachen und des Pumping Lemmas für reguläre Sprachen identisch. Wir haben bereits den Nachweis geführt, dass die obigen Sprachen aufgrund des Pumping Lemmas für reguläre Sprachen nicht regulär sind. Daher können diese Sprachen auch nicht kontextfrei sein.

Tatsächlich gilt sogar der folgende Satz:

**Satz.**
JEDE KONTEXTFREIE SPRACHE ÜBER EINEM EINELEMENTIGEN ALPHABET IST BEREITS REGULÄR.

**Beweis:** Sei $n$ die Pumping Lemma Zahl zu $L$. Jedes Wort $z \in L$ der Länge $\geq n$ lässt sich zerlegen in $uvwxy$ mit den Eigenschaften 1,2,3. Da $L \subseteq \{a\}^*$ für ein Zeichen $a$ gilt, können diese Eigenschaften einfacher formuliert werden: Es gilt $z = a^m = a^k a^l$, wobei $m \geq n$, $k+l = m$, $1 \leq l \leq n$, und $a^k a^{il} \in L$ für $i = 0, 1, 2, \ldots$. Für jedes $z \in L$, $|z| \geq n$, gibt es also derartige Zahlen $k$ und $l$. Da $l \leq n$ kommen für die Wörter $z \in L$, $|z| \geq n$, insgesamt nur endlich viele $l$-Werte vor, sagen wir $l_1, l_2, \ldots, l_p$. Sei $q \geq n$ eine Zahl, die von allen $l_i$ geteilt wird (etwa $q = n!$); und sei $q' \geq q$ eine „geeignet gewählte“ Zahl, die wir noch später bestimmen. Betrachte die Sprache

$$L' = \{x \in L \mid |x| < q\} \cup \{a^r a^{iq} \mid q \leq r \leq q',\ a^r \in L,\ i = 0, 1, 2, \ldots\}$$

Dann ist $L'$ sicherlich regulär, und es ist klar, dass $L' \subseteq L$ gilt. Wir zeigen, wenn $q'$ genügend groß ist, dann gilt auch $L \subseteq L'$. Bis zu Wörtern der Länge $< q$ stimmen $L$ und $L'$ überein. Sei nun $z = a^m \in L$, $m \geq q$. Das Wort $z$ liegt in $L'$, falls es ein Wort $a^r$ in $L$ gibt mit $q \leq r \leq q'$ und $r \equiv m \pmod{q}$. Damit ist nun alles klar: Wir wählen $q'$ so groß, dass die Wörter in $L$ mit den Längen $q, \ldots, q'$ alle möglichen Reste modulo $q$ bilden, die unter allen Wörtern in $L$ (der Länge $\geq q$) überhaupt auftreten. Da es nur endlich viele solche Reste gibt, gibt es eine solche endliche Zahl $q'$. □

### 3.3.3 Abschlusseigenschaften

Wir wollen nun untersuchen, unter welchen Operationen die kontextfreien Sprachen abgeschlossen sind (vgl. hierzu den mathematischen Anhang).

**Satz.**
DIE KONTEXTFREIEN SPRACHEN SIND ABGESCHLOSSEN UNTER:

- VEREINIGUNG
- PRODUKT
- STERN

DIE KONTEXTFREIEN SPRACHEN SIND NICHT ABGESCHLOSSEN UNTER:

- SCHNITT
- KOMPLEMENT

**Beweis:** Der Abschluss unter Vereinigung ist trivial: Falls $G_1 = (V_1, \Sigma, P_1, S_1)$ und $G_2 = (V_2, \Sigma, P_2, S_2)$, $V_1 \cap V_2 = \emptyset$ kontexfreie Grammatiken sind, so ist $G = (V_1 \cup V_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 | S_2\}, S)$ eine kontextfreie Grammatik für die Vereinigungsmenge.

Abschluss unter Produkt: Falls $G_1 = (V_1, \Sigma, P_1, S_1)$ und $G_2 = (V_2, \Sigma, P_2, S_2)$, $V_1 \cap V_2 = \emptyset$ kontexfreie Grammatiken sind, so ist $G = (V_1 \cup V_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$ eine kontextfreie Grammatik für das Produkt.

Abschluss unter Sternoperation: Falls $G_1 = (V_1, \Sigma, P_1, S_1)$ eine kontexfreie Grammatik ist (bei der o.B.d.A. $S_1$ auf keiner rechten Seite vorkommt), so ist

$$G = (V_1 \cup \{S\}, \Sigma, P_1 \cup \{S \to \varepsilon, S \to S_1,\ S_1 \to S_1 S_1\} - \{S_1 \to \varepsilon\},\ S)$$

eine kontextfreie Grammatik für die reflexive und transitive Hülle von $L(G_1)$.

Die kontextfreien Sprachen sind nicht unter Schnitt abgeschlossen. Die Sprachen

$$L_1 = \{a^i b^j c^j \mid i, j > 0\}$$

und

$$L_2 = \{a^i b^i c^j \mid i, j > 0\}$$

sind beide kontextfrei. (Zum Beispiel kann $L_1$ durch die Grammatik $S \to AB$, $A \to a|aA$, $B \to bc|bBc$ dargestellt werden). Der Schnitt von $L_1$ und $L_2$ ist jedoch die Sprache

$$\{a^i b^i c^i \mid i > 0\}$$

die bekanntermaßen nicht kontextfrei ist (vgl. Seite 93).

Deshalb können die kontextfreien Sprachen auch nicht unter Komplement abgeschlossen sein. Wenn sie es doch wären, so ließe sich der Abschluss unter Schnitt (unter Zuhilfenahme des Vereinigungsabschlusses und unter Verwendung der deMorganschen Regeln) herleiten, da

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

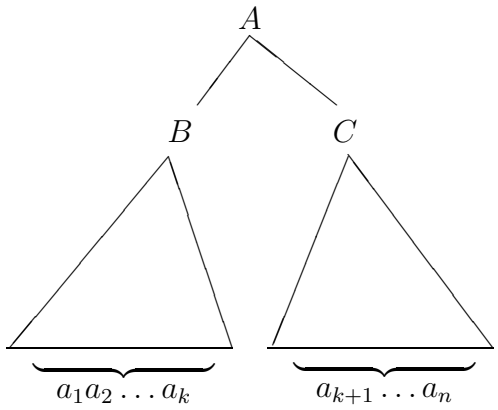
gilt. □

### 3.3.4 Der CYK-Algorithmus

Wir wissen, dass das Wortproblem für Typ 1,2,3 Grammatiken entscheidbar ist. Der entsprechende Algorithmus (vgl. Seite 62) hat allerdings – wegen seiner Allgemeinheit – exponentiellen Aufwand.

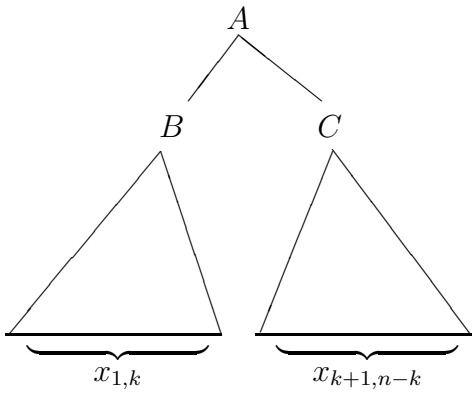
Wir werden nun einen wesentlich effizienteren Algorithmus für kontextfreie Sprachen kennenlernen – sofern diese durch Grammatiken in *Chomsky Normalform* gegeben sind. Dieser Algorithmus ist nach den Anfangsbuchstaben seiner drei Erfinder *Cocke, Younger* und *Kasami* benannt.

Wenn ein Wort $x = a$ der Länge 1 abgeleitet werden kann, so ist dies nur aufgrund einer Regel der Form $A \to a$ möglich. Gilt aber $x = a_1 a_2 \dots a_n$ mit $n \geq 2$, dann kann $x$ aus einer Variablen $A$ nur deshalb ableitbar sein, weil zunächst eine Regel der Form $A \to BC$ angewandt worden ist. Von $B$ aus wird dann ein gewisses Anfangsstück von $x$ abgeleitet und von $C$ aus das Endstück. Es muss also ein $k$ mit $1 \leq k < n$ geben, so dass gilt:




Damit ist es möglich, das Wortproblem für $x$ mit Länge $n$ auf zwei entsprechende Entscheidungen für Wörter der Länge $k$ und $n - k$ zurückzuführen. Hierbei steht $k$ jedoch nicht fest; es müssen alle Werte von 1 bis $n - 1$ in Betracht gezogen werden. Dies legt die Methode des *dynamischen Programmierens* nahe: Beginnend mit der Länge 1 untersuchen wir systematisch alle Teilwörter von $x$ auf ihre eventuelle Ableitbarkeit aus einer Variablen der Grammatik. Alle diese Informationen legen wir in einer Tabelle ab. Wenn nun ein Teilwort der Länge $m \leq n$ untersucht werden soll, so stehen die Informationen über alle kürzeren Teilwörter bereits vollständig zur Verfügung.

Die folgende Notation erweist sich als nützlich: Für ein Wort $x$ bezeichnet $x_{i,j}$ dasjenige Teilwort von $x$, das an Position $i$ beginnt und Länge $j$ hat. Mit dieser Notation sieht das obige Bild folgendermaßen aus:




Der folgende Algorithmus verwendet eine Tabelle T[1..n,1..n], wobei aber nicht alle Matrixelemente benötigt werden, sondern nur eine Dreiecksmatrix: Für $j = 1, \ldots, n$ und für $i = 1, \ldots, n + 1 - j$ wird in T[i,j] notiert, aus welchen Variablen das Wort $x_{i,j}$ abgeleitet werden kann.

Das Eingabewort $x = a_1 \ldots a_n$ ist in $L(G)$, falls sich schließlich $S \in$ T[1,n] ergibt.

**CYK-Algorithmus**

Eingabe: $x = a_1 a_2 \ldots a_n$

FOR $i := 1$ TO $n$ DO $(*$ Fall $j = 1 *)$
  $T[i, 1] := \{A \in V \mid A \to a_i \in P\}$
END;
FOR $j := 2$ TO $n$ DO $(*$ Fall $j > 1 *)$
  FOR $i := 1$ TO $n + 1 - j$ DO
    $T[i, j] := \emptyset$ ;
    FOR $k := 1$ TO $j - 1$ DO
      $T[i, j] := T[i, j] \cup \{A \mid A \to BC \in P \wedge$
        $B \in T[i, k] \wedge C \in T[i + k, j - k]\}$
    END;
  END;
END;
IF $S \in T[1, n]$ THEN
  WriteString('x liegt in L(G)')
ELSE
  WriteString('x liegt nicht in L(G)')
END

Es ist offensichtlich, dass dieser Algorithmus die Komplexität $O(n^3)$ hat, denn er besteht aus 3 ineinander verschachtelten FOR-Schleifen, die jeweils $O(n)$ viele Elemente durchlaufen.

*Beispiel:* Die Sprache

$$L = \{a^n b^n c^m \mid n, m \geq 1\}$$

ist kontextfrei:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow ab \mid aAb \\ B &\rightarrow c \mid cB \end{aligned}$$

Umformen in CNF ergibt:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow CD \mid CF \\ B &\rightarrow c \mid EB \\ C &\rightarrow a \\ D &\rightarrow b \\ E &\rightarrow c \\ F &\rightarrow AD \end{aligned}$$

Sei $x = aaabbbcc$. Dann erzeugt der Algorithmus die folgende Tabelle:

| | $i \rightarrow$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $x =$ | $a$ | $a$ | $a$ | $b$ | $b$ | $b$ | $c$ | $c$ |
| $j$ $\downarrow$ | $C$ | $C$ | $C$ | $D$ | $D$ | $D$ | $E, B$ | $E, B$ |
| | | | $A$ | | | | $B$ | |
| | | | $F$ | | | | | |
| | | $A$ | | | | | | |
| | | $F$ | | | | | | |
| | $A$ | | | | | | | |
| | $S$ | | | | | | | |
| | $S$ | | | | | | | |

Da $S$ im untersten Kästchen vorkommt, liegt $x$ in der Sprache.

### 3.3.5 Kellerautomaten

Wir wollen nun das Modell des endlichen Automaten so erweitern, dass dieses neue Automatenmodell genau die kontextfreien Sprachen erkennt. Den endlichen Automaten mangelte es an irgendeiner Form eines *Speichers*. Intuitiv kann ein endlicher Automat eine

Sprache wie

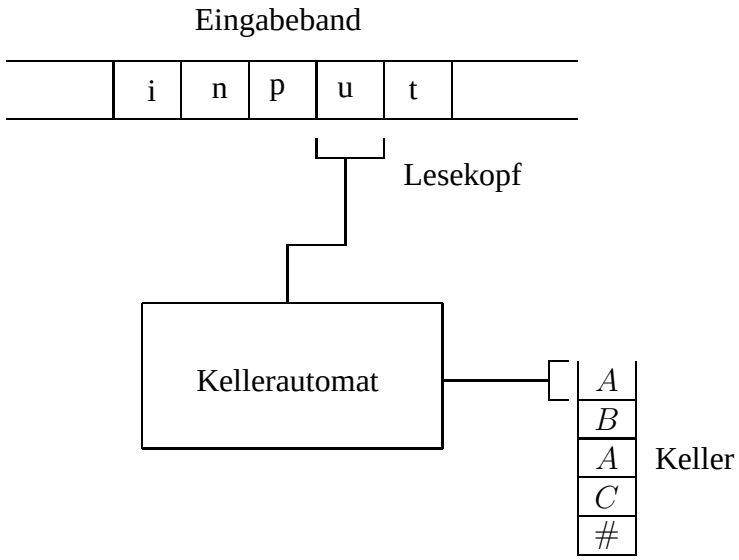
$$L = \{a_1 a_2 \dots a_n \$ a_n \dots a_2 a_1 \mid a_i \in \Sigma\}$$

deshalb nicht erkennen, weil er zum Zeitpunkt, wenn er das Eingabezeichen \$ erreicht, nicht mehr „wissen“ kann, was $a_1 a_2 \dots a_n$ war. Die einzige „gespeicherte Information“, die ihm zur Verfügung steht, ist der Zustand, in dem er sich befindet (und davon gibt es nur endlich viele).

Beim Kellerautomaten wird das NFA-Modell um einen Speicher erweitert, auf den jedoch nur in der Art eines *Kellers*, eines *pushdown*-Speichers zugegriffen werden kann. (Man beachte, ein Kellerautomat ist nach Definition zunächst *nichtdeterministisch*).

Die möglichen Aktionen eines Kellerautomaten hängen jetzt nicht nur vom Zustand und gelesenen Eingabezeichen ab, sondern auch vom Kellerinhalt (bzw. dem zur Zeit obersten Kellerzeichen). In jeden „Rechenschritt“ kann sich nicht nur der Zustand, sondern auch der Inhalt des Kellers verändern.

*Skizze:*




**Definition.** Ein *(nichtdeterministischer) Kellerautomat* (engl.: *pushdown automaton*, kurz: PDA) wird angegeben durch ein 6-Tupel

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$$

Hierbei sind:

- $Z$ die endliche Menge der *Zustände*
- $\Sigma$ das *Eingabealphabet*
- $\Gamma$ das *Kelleralphabet*

- $\delta : Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow \mathcal{P}_e(Z \times \Gamma^*)$ die *Überführungsfunktion* (Hierbei bedeutet $\mathcal{P}_e$ die Menge aller *endlichen* Teilmengen)
- $z_0 \in Z$ der *Startzustand*
- $\# \in \Gamma$ das *unterste Kellerzeichen*

Intuitiv bedeutet

$$\delta(z, a, A) \ni (z', B_1 \dots B_k)$$

folgendes: Wenn sich $M$ im Zustand $z$ befindet, das Eingabezeichen $a$ liest, und $A$ das oberste Kellerzeichen ist, so *kann* $M$ im nächsten Schritt in den Zustand $z'$ übergehen und das oberste Kellerzeichen $A$ durch die Zeichen $B_1 \dots B_k$ ersetzen. (Danach ist $B_1$ das oberste Kellerzeichen).

Man beachte, dass dies den Fall miteinschließt, dass $A$ entfernt wird (die POP-Operation), wenn $k = 0$ gewählt wird. Es ist zum Beispiel auch möglich, dass, ohne $A$ zu verändern, ein weiteres Zeichen $B$ in den Keller „gePUSHt“ wird ($B_1 \dots B_k = BA$).

In der Definition von $\delta$ ist auch zugelassen, dass an der Stelle eines Eingabezeichens $a \in \Sigma$ das leere Wort $\varepsilon$ steht. In diesem Fall findet ein sog. *spontaner Übergang*, ohne Lesen eines Eingabezeichens, statt.

Intuitiv bedeutet

$$\delta(z, \varepsilon, A) \ni (z', B_1 \dots B_k)$$

folgendes: Wenn sich $M$ im Zustand $z$ befindet und $A$ das oberste Kellerzeichen ist, so *kann* $M$ im nächsten Schritt – ohne Lesen eines Eingabezeichens – in den Zustand $z'$ übergehen und das oberste Kellerzeichen $A$ durch die Zeichen $B_1 \dots B_k$ ersetzen. (Danach ist $B_1$ das oberste Kellerzeichen).

Was das Akzeptieren eines Eingabewortes betrifft, so weichen wir beim PDA insofern vom endlichen Automaten ab, dass es keine Endzustände gibt. Stattdessen werden akzeptierte Wörter dadurch charakterisiert, dass der Keller nach Abarbeiten eines solchen Wortes leer ist. (Zu Beginn jeder Rechnung steht immer das Zeichen $\#$ im Keller).

Tatsächlich kann man zeigen, dass diese Form des *Akzeptierens durch leeren Keller* mit der des *Akzeptierens durch Endzustand* gleichwertig sind. Für das Folgende ist das Akzeptieren durch leeren Keller jedoch die praktischere Version.

Wir haben – im Unterschied zu NFAs – aus Einfachheitsgründen nur einen einzigen Startzustand in der Definition zugelassen. Dies ist keine echte Einschränkung, da mittels spontaner Übergänge von $z_0$ aus jeder mögliche „eigentliche“ Startzustand, ohne Lesen eines Eingabezeichens, erreichbar ist.

> **Definition.** Eine *Konfiguration* $k$ eines Kellerautomaten ist gegeben durch ein Tripel $k \in Z \times \Sigma^* \times \Gamma^*$.
>
> (Die Idee hierbei ist, dass durch ein Konfigurationstripel eindeutig eine „Momentaufnahme“ des Kellerautomaten beschrieben wird; und zwar durch Angabe des momentanen Zustands, des noch zu lesenden Teils der Eingabe und

des aktuellen Kellerinhalts – das oberste Kellerzeichen hierbei ganz links stehend).

Auf der Menge aller Konfigurationen definieren wir eine zweistellige Relation $\vdash$ wie folgt. Informal gesprochen soll $k \vdash k'$ genau dann gelten, wenn die Konfiguration $k'$ aus $k$ in einem „Rechenschritt“ (=eine Anwendung der $\delta$-Funktion) hervorgeht.

Formal ausgedrückt:

$$(z, a_1 \dots a_n, A_1 \dots A_m) \vdash \begin{cases} (z', a_2 \dots a_n, B_1 \dots B_k A_2 \dots A_m), \\ \quad \text{falls } \delta(z, a_1, A_1) \ni (z', B_1 \dots B_k) \\ (z', a_1 a_2 \dots a_n, B_1 \dots B_k A_2 \dots A_m) \\ \quad \text{falls } \delta(z, \varepsilon, A_1) \ni (z', B_1 \dots B_k) \end{cases}$$

Sei $\vdash^*$ die reflexive und transitive Hülle von $\vdash$ (siehe mathematischen Anhang). Die durch einen Kellerautomaten $M$ *(durch leeren Keller) akzeptierte Sprache* ist

$$N(M) = \{x \in \Sigma^* \mid (z_0, x, \#) \vdash^* (z, \varepsilon, \varepsilon) \text{ für ein } z \in Z\}$$

*Beispiel:* Wir wollen einen Kellerautomaten für die obige Beispielsprache

$$L = \{a_1 a_2 \dots a_n \$ a_n \dots a_2 a_1 \mid a_i \in \{a, b\}\}$$

angeben. Sei

$$M = (\{z_0, z_1\}, \{a, b, \$\}, \{\#, A, B\}, \delta, z_0, \#)$$

Um Schreibarbeit zu sparen, schreiben wir statt $\delta(z, a, A) \ni (z', x)$ einfach $zaA \to z'x$:

$$\begin{array}{lll} z_0 a\# \to z_0 A\#, & z_0 aA \to z_0 AA, & z_0 aB \to z_0 AB \\ z_0 b\# \to z_0 B\#, & z_0 bA \to z_0 BA, & z_0 bB \to z_0 BB \\ z_0 \$\# \to z_1 \#, & z_0 \$A \to z_1 A, & z_0 \$B \to z_1 B \\ z_1 aA \to z_1 \varepsilon, & z_1 bB \to z_1 \varepsilon, & z_1 \varepsilon\# \to z_1 \varepsilon \end{array}$$

Es gilt zum Beispiel $ba\$ab \in N(M)$, denn:

$$\begin{array}{l} (z_0, ba\$ab, \#) \vdash (z_0, a\$ab, B\#) \vdash (z_0, \$ab, AB\#) \vdash \\ (z_1, ab, AB\#) \vdash (z_1, b, B\#) \vdash (z_1, \varepsilon, \#) \vdash (z_1, \varepsilon, \varepsilon) \end{array}$$

Man erkennt, dass bei diesem Kellerautomaten jede Konfiguration nur eine Folgekonfiguration besitzt: der Automat ist sogar deterministisch.

Ersetzt man die 3. Zeile der Definition von $\delta$ durch die Übergänge

$$z_0 aA \to z_1 \varepsilon, \quad z_0 bB \to z_1 \varepsilon, \quad z_0 \varepsilon\# \to z_1 \varepsilon$$

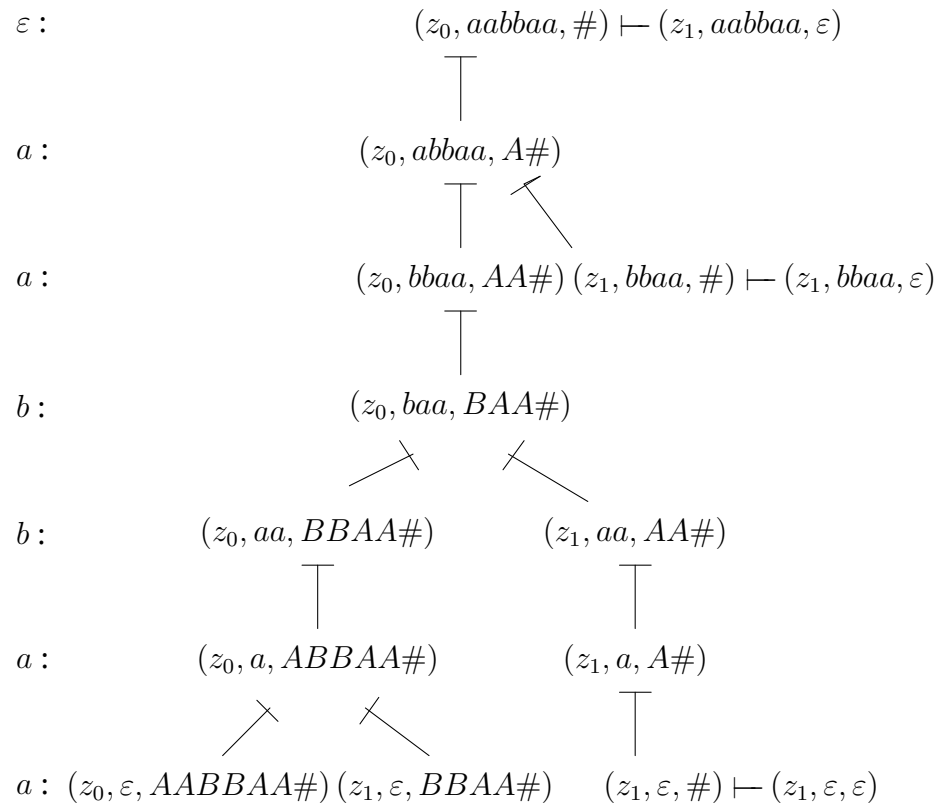
so erhält man einen Kellerautomaten $M'$, für den

$$N(M') = \{a_1 a_2 \dots a_n a_n \dots a_2 a_1 \mid a_i \in \{a, b\}\}$$

ist. $M'$ hat die möglichen Übergänge


Beim Abarbeiten von $aabbaa$ sind folgende Konfigurationsfolgen möglich:

*Eingabezeichen:* *Konfigurationen:*




Hier wird der Nichtdeterminismus benötigt, um die Wortmitte zu „erraten“. Man kann zeigen, dass es für die von $M'$ akzeptierte Sprache keinen äquivalenten *deterministischen* Kellerautomaten gibt.

**Satz.**
EINE SPRACHE $L$ IST KONTEXTFREI GENAU DANN, WENN $L$ VON EINEM NICHTDETERMINISTISCHEN KELLERAUTOMATEN ERKANNT WIRD.

**Beweis:** (⇒) Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik für $L$. Wir geben einen Kellerautomaten $M$ an, der Ableitungen von $G$ mit seinem Kellerinhalt simuliert. Hierbei wählen wir als Kelleralphabet $\Gamma = V \cup \Sigma$ und die Startvariable $S$ als das unterste

Kellerzeichen. Sobald dieses gePOPt wird, bedeutet das, dass eine Ableitung für das Eingabewort gefunden wurde.

$$M = (\{z\}, \Sigma, V \cup \Sigma, \delta, z, S)$$

Mit Hilfe von $P$ definieren wir $\delta$ wie folgt. Für jede Regel $A \to \alpha \in P$ mit $\alpha \in (V \cup \Sigma)^*$ setze:

$$\delta(z, \varepsilon, A) \ni (z, \alpha)$$

Ferner setze:

$$\delta(z, a, a) \ni (z, \varepsilon)$$

Das heißt: immer wenn das oberste Kellerzeichen eine Variable der Grammatik ist, wird ohne Lesen eines Eingabezeichens eine $P$-Regel angewandt; immer wenn das oberste Kellerzeichen ein Terminalzeichen ist und mit dem Eingabezeichen übereinstimmt, wird dieses einfach vom Keller gePOPt.

Es gilt nun für alle $x \in \Sigma^*$:

$x \in L(G)$

**genau dann wenn** es gibt eine Ableitung in $G$ der Form $S \Rightarrow \cdots \Rightarrow x$

**genau dann wenn** es gibt eine Folge von Konfigurationen von $M$ der Form $(z, x, S) \vdash \cdots \vdash (z, \varepsilon, \varepsilon)$

**genau dann wenn** $x \in N(M)$.

($\Leftarrow$) Sei $L = N(M)$ für einen Kellerautomaten $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$. Wir können annehmen, dass für jede $\delta$-Regel $zaA \to z'B_1 \dots B_k$ gilt: $k \leq 2$. Denn komme etwa $zaA \to z'B_1 \dots B_k$ in $\delta$ vor mit $k > 2$, so können wir neue Zustände $z_1, \dots, z_{k-2}$ wählen und diese $\delta$-Regel ersetzen durch:

$$\begin{aligned} zaA &\to z_1 B_{k-1} B_k \\ z_1 \varepsilon B_{k-1} &\to z_2 B_{k-2} B_{k-1} \\ &\vdots \\ z_{k-2} \varepsilon B_2 &\to z' B_1 B_2 \end{aligned}$$

Wir konstruieren nun eine Grammatik $G$, die Rechenschritte von $M$ durch Linksableitungsschritte simuliert. Die „Variablen“ dieser Grammatik setzen sich aus mehreren Bestandteilen zusammen (Kreuzprodukt), nämlich dem Zustand vor einer Folge von Rechenschritten des Kellerautomaten, dem verarbeiteten Kellersymbol und dem Zustand, der durchlaufen wird, wenn dieses Kellerzeichen wieder gePOPt wird.

Sei

$$G = (V, \Sigma, P, S)$$

wobei

$$V = \{S\} \cup Z \times \Gamma \times Z$$

und

$$\begin{aligned} P \;&= \{S \to (z_0, \#, z) \mid z \in Z\} \\ &\cup \{(z, A, z') \to a \mid \delta(z, a, A) \ni (z', \varepsilon)\} \\ &\cup \{(z, A, z') \to a(z_1, B, z') \mid \delta(z, a, A) \ni (z_1, B),\ z' \in Z\} \\ &\cup \{(z, A, z') \to a(z_1, B, z_2)(z_2, C, z') \mid \delta(z, a, A) \ni (z_1, BC), \\ &\qquad z', z_2 \in Z\} \end{aligned}$$

Diese Grammatik kann auch $\varepsilon$-Produktionen enthalten (denn $a$ kann auch $\varepsilon$ sein). Diese können nach dem Verfahren auf Seite 59 noch eliminiert werden.

Wir beweisen nun für alle $x \in \Sigma^*$ die folgende Behauptung:

$$\boxed{(z, A, z') \Rightarrow^* x \;\textbf{ genau dann wenn }\; (z, x, A) \vdash^* (z', \varepsilon, \varepsilon)}$$

Für $a \in \Sigma \cup \{\varepsilon\}$ beobachten wir zunächst:

$$\begin{aligned} (z, A, z') \Rightarrow a \quad &\textbf{gdw} \quad (z, A, z') \to a \in P \\ &\textbf{gdw} \quad \delta(z, a, A) \ni (z', \varepsilon) \\ &\textbf{gdw} \quad (z, a, A) \vdash (z', \varepsilon, \varepsilon) \end{aligned}$$

Wir zeigen nun die Richtung von rechts nach links durch Induktion über die Anzahl $n$ der Rechenschritte von $M$. Der kürzest-mögliche Fall ist $n = 1$. Dieser wird durch obige Beobachtung abgehandelt.

Sei nun $n > 1$, dann hat $x$ die Form $x = ay$, $a \in \Sigma \cup \{\varepsilon\}$, so dass gilt: $(z, ay, A) \vdash (z_1, y, \alpha) \vdash^+ (z', \varepsilon, \varepsilon)$ für einen gewissen Zustand $z_1$ und Kellerinhalt $\alpha$. Wir unterscheiden nun die drei denkbaren Fälle $\alpha = \varepsilon$, $\alpha = B$ und $\alpha = BC$.

Der *Fall* $\alpha = \varepsilon$ ist nicht möglich, da $(z_1, y, \varepsilon)$ keine Folgekonfiguration besitzt.

*Fall* $\alpha = B$ : Dann gilt $(z_1, B, z') \Rightarrow^* y$ nach Induktionsvoraussetzung. Außerdem muss es in $P$ eine Regel der Form $(z, A, z') \to a(z_1, B, z')$ geben (dies ergibt sich aus der Form des ersten Rechenschritts). Damit erhalten wir insgesamt: $(z, A, z') \Rightarrow a(z_1, B, z') \Rightarrow^* ay = x$.

*Fall* $\alpha = BC$ : Die Konfigurationsfolge $(z_1, y, BC) \vdash^* (z', \varepsilon, \varepsilon)$ kann in zwei Teile zerlegt werden: $(z_1, y, BC) \vdash^* (z_2, y_2, C)$ und $(z_2, y_2, C) \vdash^* (z', \varepsilon, \varepsilon)$, so dass $y_2$ ein gewisses Endstück von $y$ ist, d.h. $y = y_1 y_2$. Für $y_1$, den vorderen Teil von $y$, gilt ferner: $(z_1, y_1, B) \vdash^* (z_2, \varepsilon, \varepsilon)$. Nach Induktionsvoraussetzung gilt sowohl $(z_1, B, z_2) \Rightarrow^* y_1$ als auch $(z_2, C, z') \Rightarrow^* y_2$. Ferner muss es in $P$ eine Regel der Form $(z, A, z') \to a(z_1, B, z_2)(z_2, C, z')$ geben (dies ergibt sich aus der Form des ersten Rechenschritts). Zusammengefasst erhalten wir:

$$(z, A, z') \Rightarrow a(z_1, B, z_2)(z_2, C, z') \Rightarrow^* ay_1(z_2, C, z') \Rightarrow^* ay_1y_2 = x$$

Die Richtung von links nach rechts zeigen wir durch Induktion nach $k$, der Länge der Linksableitung von $x$.

Der Induktionsanfang ($k = 1$) ergibt sich wieder aus der obigen Beobachtung.

Betrachten wir nun eine Ableitung mit $k > 1$.

*Fall 1*: $(z, A, z') \Rightarrow a \Rightarrow^* x$. Dann ist $x = a$. Dies ist bei $k > 1$ nicht möglich.

*Fall 2*: $(z, A, z') \Rightarrow a(z_1, B, z') \Rightarrow^* ay = x$. Dann ist $\delta(z, a, A) \ni (z_1, B)$ und nach Induktionsvoraussetzung gilt $(z_1, y, B) \vdash^* (z', \varepsilon, \varepsilon)$. Daraus folgt $(z, ay, A) \vdash (z_1, y, B) \vdash^* (z', \varepsilon, \varepsilon)$.

*Fall 3*: $(z, A, z') \Rightarrow a(z_1, B, z_2)(z_2, C, z') \Rightarrow^* ay = x$. Dann ist $\delta(z, a, A) \ni (z_1, BC)$ und nach Induktionsvoraussetzung gilt $(z_1, y_1, B) \vdash^* (z_2, \varepsilon, \varepsilon)$ und $(z_2, y_2, C) \vdash^* (z', \varepsilon, \varepsilon)$ wobei $y = y_1 y_2$. Daraus folgt

$$(z, ay_1y_2, A) \vdash (z_1, y_1y_2, BC) \vdash^* (z_2, y_2, C) \vdash^* (z', \varepsilon, \varepsilon)$$

Mit dieser Behauptung ergibt sich $N(M) = L(G)$ wie folgt:

$$\begin{array}{lll} x \in N(M) & \textbf{gdw} & (z_0, x, \#) \vdash^* (z, \varepsilon, \varepsilon) \text{ für ein } z \in Z \\ & \textbf{gdw} & S \Rightarrow (z_0, \#, z) \Rightarrow^* x \text{ für ein } z \in Z \\ & \textbf{gdw} & x \in L(G). \end{array}$$

□

*Bemerkung:* Der Beweis des vorigen Satzes zeigt, dass man jeden Kellerautomaten so umkonstruieren kann, dass er nur *einen* Zustand hat: Denn sei $M$ ein beliebiger Kellerautomat. Dann gilt aufgrund des Satzes $N(M) = L(G)$ für eine kontextfreie Grammatik $G$. Wenn wir den Satz nun wieder auf $G$ anwenden, so erhalten wir einen äquivalenten Kellerautomaten, der mit einem Zustand auskommt.

*Bemerkung:* Da jede kontextfreie Grammatik (ohne $\varepsilon$) in Greibach Normalform umgeformt werden kann (vgl. Seite 90), kann der erste Teil des Beweises auch so geführt werden, dass für jede Regel $A \to aB_1 \dots B_k$ gilt $\delta(z, a, A) \ni (z, B_1 \dots B_k)$. Das heißt, dass man immer einen Kellerautomaten angeben kann, der in jedem Schritt ein Eingabezeichen abliest und keine spontanen Übergänge hat.

### 3.3.6 Deterministisch kontextfreie Sprachen

Kellerautomaten wurden zunächst als nichtdeterministisches Konzept eingeführt – und nur in dieser Form sind die von ihnen erkannten Sprachen *genau* die kontextfreien Sprachen.

Nun sollen auch deterministische Kellerautomaten eingeführt werden. Wir werden sehen, dass diese nur eine *echte Teilmenge* der kontextfreien Sprachen definieren – allerdings auch nach wie vor eine *echte Obermenge* der regulären Sprachen. Wir wollen bei der Definition von deterministischen Kellerautomaten (engl. kurz: DPDA) das Konzept des

spontanen $\varepsilon$-Übergangs beibehalten. Deshalb muss man bei der folgenden Definition dies mitberücksichtigen.

> **Definition.** Ein Kellerautomat $M$ heißt *deterministisch*, falls für alle $z \in Z,\ a \in \Sigma$ und $A \in \Gamma$ gilt:
>
> $$|\delta(z, a, A)| + |\delta(z, \varepsilon, A)| \leq 1$$
>
> Es kommt hinzu, dass deterministisch kontextfreie Kellerautomaten *per Endzustand* akzeptieren und nicht *per leerem Keller*.[1]
>
> Eine Sprache heißt *deterministisch kontextfrei*, falls sie von einem deterministischen Kellerautomaten erkannt wird.

*Beispiel:* Die Sprache

$$L = \{a_1 \dots a_n \$ a_n \dots a_1 \mid a_i \in \Sigma\}$$

ist deterministisch kontextfrei, nicht jedoch

$$L' = \{a_1 \dots a_n a_n \dots a_1 \mid a_i \in \Sigma\}.$$

Aus der Definition der deterministisch kontextfreien Sprachen ergibt sich, dass Konfigurationsbäume zu linearen Ketten „degenerieren“. Das heißt, die Relation $\vdash$ wird hier zu einer Funktion, d. h. für jede Konfiguration $k$ gibt es höchstens eine Konfiguration $k'$ mit $k \vdash k'$:

$$(z_0, x, \#) \vdash k_1 \vdash k_2 \vdash \cdots \vdash k_n \vdash \cdots$$

Wir erwähnen ohne Beweis den folgenden

**Satz.**
DIE DETERMINISTISCH KONTEXTFREIEN SPRACHEN SIND UNTER KOMPLEMENTBILDUNG ABGESCHLOSSEN.

Die kontextfreien Sprachen sind – wie bekannt – nicht Schnitt-abgeschlossen. Das angegebene Gegenbeispiel waren die Sprachen

$$L_1 = \{a^n b^n c^m \mid n, m \geq 1\}$$

und

$$L_2 = \{a^n b^m c^m \mid n, m \geq 1\}$$

[1] Tatsächlich ist dies für *deterministische* Kellerautomaten ein Unterschied; für nichtdeterministische sind beide Akzeptiermechanismen äquivalent.

Man stellt fest, dass diese Sprachen sogar *deterministisch* kontextfrei sind. Daher sind die deterministisch kontextfreien Sprachen gleichfalls nicht unter Schnitt abgeschlossen.

Dann können sie aber auch nicht unter Vereinigung abgeschlossen sein: Wenn dies der Fall wäre, ließe sich der Schnitt mittels Vereinigung und Komplement nach der deMorgan'schen Regel

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

darstellen.

Wir fassen zusammen:

**Satz.**
DIE DETERMINISTISCH KONTEXTFREIEN SPRACHEN SIND *nicht* UNTER SCHNITT UND VEREINIGUNG ABGESCHLOSSEN.

*Bemerkung:* Die deterministisch kontextfreien Sprachen stimmen mit den sog. $LR(k)$-Sprachen überein. Diese spielen im Compilerbau eine wichtige Rolle ($\rightarrow$ Vorlesung über Compilerbau oder Syntaxanalyse).

Für deterministisch kontextfreie Sprachen ist das Wortproblem in *linearer Zeit* lösbar.

**Satz.**
DER SCHNITT EINER (DETERMINISTISCH) KONTEXTFREIEN SPRACHE MIT EINER REGULÄREN SPRACHE IST WIEDER (DETERMINISTISCH) KONTEXTFREI.

**Beweis:** Sei $M_1 = (Z_1, \Sigma, \Gamma, \delta_1, z_{01}, \#, E_1)$ ein Kellerautomat (mit Endzustandsmenge $E_1$) für die Sprache $L_1$ und sei $M_2 = (Z_2, \Sigma, \delta_2, z_{02}, E)$ ein DFA für die Sprache $L_2$. Ähnlich der Konstruktion des „Kreuzproduktautomaten“ auf Seite 85 definieren wir einen Kellerautomaten $M_3$ (desselben Typs wie $M_1$), der die Sprache $L_1 \cap L_2$ erkennt. Es ist

$$M_3 = (Z_1 \times Z_2, \Sigma, \Gamma, \delta_3, (z_{01}, z_{02}), \#, E_1 \times E_2)$$

wobei $\delta_3((z_1, z_2), a, A) \ni ((z_1', z_2'), B_1 \dots B_k)$, falls $\delta_1(z_1, a, A) \ni (z_1', B_1 \dots B_k)$ und $\delta_2(z_2, a) = z_2'$. □

### 3.3.7 Entscheidbarkeit

Wir haben bereits mit dem CYK-Algorithmus einen effizienten Algorithmus für das Wortproblem bei kontextfreien Grammatiken vorgestellt. Insbesondere heißt dies, dass das Problem, gegeben eine kontextfreie Grammatik $G$ und ein Wort $x$, festzustellen, ob $x \in L(G)$, *entscheidbar* ist. Entscheidbarkeit bedeutet (intuitiv), dass es einen Algorithmus gibt, der immer stoppt, und das betreffende zu lsende Problem korrekt lst.

Ein weiteres entscheidbares Problem ist das *Leerheitsproblem*. Hierzu gehen wir aus von einer kontextfreien Grammatik in CNF. Wir markieren alle Variablen, die in der Lage sind, Terminalwörter abzuleiten. Hierzu markieren wir zunächst alle Variablen $A$, sofern

$A \to a$ eine Regel ist. Als nächstes markieren wir sukzessive alle Variablen $A$, sofern $A \to BC$ eine Regel ist und $B$ und $C$ bereits markiert sind. Offensichtlich ist die erzeugte Sprache genau dann leer, wenn bei diesem Prozess die Startvariable $S$ nicht markiert wird. Unter Verwenden derselben Idee kann man auch einen Markierungsalgorithmus angeben, der ohne zuvorige Umformung in CNF auskommt.

Entscheidbar ist ebenfalls das *Endlichkeitsproblem*. Dies sieht man sofort mit Hilfe des Pumping Lemmas: Sei $n$ die der Sprache $L$ zugeordnete Pumping Lemma Zahl. Es ist $|L| = \infty$ genau dann, wenn es ein Wort $z$ in $L$ mit $n \leq |z| < 2n$ gibt (und diese endlich vielen Wörter können auf Mitgliedschaft in $L$ getestet werden). Zur Begründung: Wenn $L$ mindestens ein Wort der Länge $\geq n$ enthält, so enthält $L$ gemäß des Pumping Lemmas unendlich viele Wörter. Sei umgekehrt $|L| = \infty$ und sei $z \geq n$ ein Wort in $L$ minimaler Länge. Wenn $|z| \geq 2n$, so kann $z$ gemäß des Pumping Lemmas zerlegt werden in $uvwxy$, so dass insbesondere $uwy$ in $L$ liegt, wobei $|uwy| \geq n$. Dies ist ein Widerspruch zur Minimalität von $z$. Daher muss $L$ ein Wort der Länge $\geq n$ und $< 2n$ enthalten.

Unter Effizienzaspekten ist der zuletzt angegebene Algorithmus jedoch nicht empfehlenswert, denn es müssen exponentiell in $n$ viele Wörter getestet werden. (Hinzu kommt noch, dass bereits $n$ exponentiell in $|V|$ ist). Effizientere Verfahren suchen nach bestimmten Zyklen in dem „Grammatik-Graphen“ mit der Knotenmenge $V$, den man einer kontextfreien Grammatik zuordnen kann.

Ein weiteres entscheidbares Problem ist das folgende: Gegeben eine deterministisch kontextfreie Sprache $L_1$ (gegeben in Form eines deterministischen Kellerautomaten), und eine reguläre Sprache $L_2$ (in Form eines DFA), stelle fest, ob $L_1 = L_2$. Die Entscheidbarkeit dieses Problems sieht man wie folgt: Es gilt $L_1 = L_2$ genau dann, wenn $L_1 \subseteq L_2$ und $L_2 \subseteq L_1$. Dies wiederum gilt genau dann, wenn $L_1 \cap \overline{L_2} = \emptyset$ und $\overline{L_1} \cap L_2 = \emptyset$. Da man effektiv zu einem gegebenen deterministischen Kellerautomaten (bzw. zu einem DFA) einen entsprechenden Automaten konstruieren kann, der das Komplement erkennt, und da man ferner zu einem deterministischen Kellerautomaten und einem DFA einen deterministischen Kellerautomaten konstruieren kann, der die Schnittmenge der beiden Sprachen erkennt (siehe Seite 107), läuft es darauf hinaus, von zwei kontextfreien Sprachen festzustellen, ob sie leer sind, und dies ist entscheidbar.

Mehr noch, tatsächlich ist sogar das Äquivalenzproblem für deterministisch kontextfreie Sprachen entscheidbar.

Fast alle Fragestellungen bei kontextfreien Sprachen, sofern sie nicht in diesem Abschnitt angesprochen wurden, sind jedoch unentscheidbar.

## 3.4 Kontextsensitive und Typ 0-Sprachen

Für die Typ 1 Grammatiken war die Bedingung, dass die rechten Regelseiten nicht kürzer sind als die linken. Für diese Grammatiken lässt sich eine Normalform angeben, die in

gewisser Weise mit der Chomsky Normalform bei den kontextfreien Grammatiken vergleichbar ist.

> **Definition.** Eine Typ 1 Grammatik ist in *Kuroda Normalform*, falls alle Regeln eine der 4 Formen haben:
>
> $$A \to a, \quad A \to B, \quad A \to BC, \quad AB \to CD.$$
>
> Hierbei stehen $A, B, C, D$ für Variablen und $a$ für ein Terminalsymbol.

**Satz.**
FÜR JEDE TYP 1 GRAMMATIK $G$ MIT $\varepsilon \notin L(G)$ GIBT ES EINE GRAMMATIK $G'$ IN KURODA NORMALFORM MIT $L(G) = L(G')$.

**Beweis:** Analog der Umformung in CNF bei kontextfreien Sprachen können wir davon ausgehen, dass alle Regeln, die Terminalzeichen involvieren, nur die Form $A \to a$ haben (vgl. Seite 89). Dies erreichen wir, indem wir für das Terminalzeichen $a$ eine neue Variable $A$ und die Regel $A \to a$ hinzufügen und $a$ in allen anderen Regeln (sowohl auf der linken wie auf der rechten Seite) durch $A$ ersetzen.

Alle Regeln der Form $A \to B_1 B_2 \dots B_k$, $k > 2$, können wie bei der Umformung in CNF in Regeln der Form $A \to BC$ aufgebrochen werden.

Die einzigen noch möglichen Regeln nach diesen Umformungsschritten, die *nicht* der Kuroda Normalform entsprechen, haben die Form $A_1 \dots A_m \to B_1 \dots B_n$, $2 \le m \le n$, wobei $m$ und $n$ nicht beide gleich 2 sind. Eine solche Regel kann ersetzt werden durch den folgenden Satz von Regeln

$$\begin{array}{rcl@{\qquad}rcl} A_1A_2 & \to & B_1C_2 & C_m & \to & B_mC_{m+1} \\ C_2A_3 & \to & B_2C_3 & C_{m+1} & \to & B_{m+1}C_{m+2} \\ \vdots & & & \vdots & & \\ C_{m-1}A_m & \to & B_{m-1}C_m & C_{n-1} & \to & B_{n-1}B_n \end{array}$$

wobei $C_2, \dots, C_{n-1}$ neue Variablen sind. Nachdem jede solche Regel durch diesen Satz von Regeln ersetzt wurde, ist die Grammatik in Kuroda Normalform. □

Als nächstes wollen wir die Typ 1 und Typ 0 Sprachen gemeinsam behandeln. Gesucht ist ein Automatenmodell, das diese Sprachtypen beschreiben kann. Es muss offensichtlich allgemeiner sein als der Kellerautomat. Die wesentliche Beschränkung des Kellerautomaten ist die Zugriffsmöglichkeit auf seinen Speicher. Er darf eben nur nach dem Kellerprinzip (Last-in, First-out) angesprochen werden.

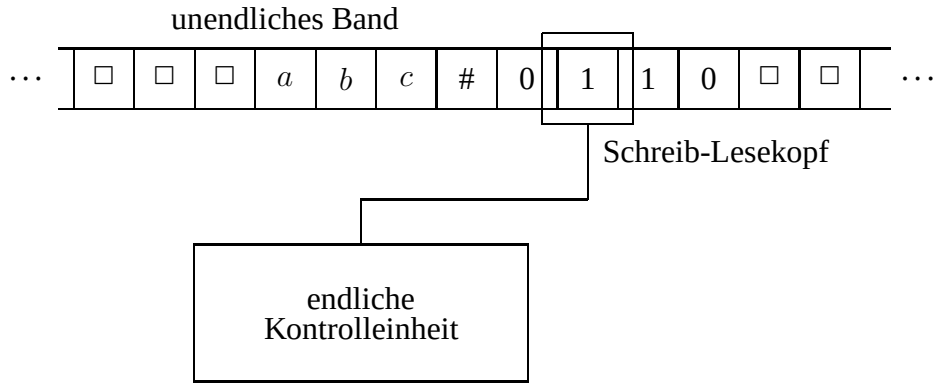
A.M. Turing[2] schlägt ein Automatenmodell vor, das „berechnungsstärker“ ist als der Kellerautomat und heute *Turingmaschine* genannt wird. Turings Intention war noch viel weit-

[2] Alan M. Turing, 1912–1954, englischer Mathematiker, Kryptoanalytiker und Computerkonstrukteur. Seine 1937 erschienene Arbeit „On computable numbers, with an application to the Entscheidungsproblem“ hat die Berechenbarkeitstheorie, und damit die Theorie der Informatik, begründet.

reichender, nämlich eine mathematisch klar beschreibbare Maschine anzugeben, die allgemein genug ist, um stellvertretend für *jeden beliebigen* algorithmischen Berechnungsprozess zu stehen.

Das heißt, Turings Vorstellung ist es, mit der Turingmaschine den (zunächst nur intuitiv gegebenen) Begriff der *Berechenbarkeit*, des *effektiven Verfahrens* exakt beschrieben zu haben. Man ist heute davon überzeugt, dass ihm dieses geglückt ist.

Anschaulich beschrieben besteht eine Turingmaschine aus einem (potenziell) unendlichen Band, das in Felder eingeteilt ist. Jedes Feld kann ein einzelnes Zeichen des sog. Arbeitsalphabets der Maschine enthalten. Auf dem Band kann sich ein Schreib-Lesekopf bewegen. Nur solche Zeichen, auf denen sich dieser Kopf gerade befindet, können in dem momentanen Rechenschritt verändert werden. Der Kopf kann in einem Rechenschritt dann um maximal eine Position nach links oder nach rechts bewegt werden.




Bandfelder, die von dem Schreib-Lesekopf noch nie besucht und verändert wurden, enthalten das „Blank"-Zeichen □.

**Definition.** Eine *Turingmaschine* (kurz: TM) ist gegeben durch ein 7-Tupel

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$$

Hierbei sind:

- $Z$ die endliche *Zustandsmenge,*
- $\Sigma$ das *Eingabealphabet,*
- $\Gamma \supset \Sigma$ das *Arbeitsalphabet,*
- $\delta : Z \times \Gamma \longrightarrow Z \times \Gamma \times \{L, R, N\}$ im deterministischen Fall (bzw. $\delta : Z \times \Gamma \longrightarrow \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$ im nichtdeterministischen Fall) die *Überführungsfunktion,*
- $z_0 \in Z$ der *Startzustand,*
- $\square \in \Gamma - \Sigma$ das *Blank,*

- $E \subseteq Z$ die Menge der *Endzustände*.

Informal erklärt bedeutet

$$\delta(z, a) = (z', b, x) \text{ bzw. } \delta(z, a) \ni (z', b, x)$$

folgendes:

Wenn sich $M$ im Zustand $z$ befindet und unter dem Schreib-Lesekopf das Zeichen $a$ steht, so geht $M$ im nächsten Schritt in den Zustand $z'$ über, schreibt (auf den Platz von $a$) $b$ auf das Band und führt danach die Kopfbewegung $x \in \{L, R, N\}$ aus. (Hierbei steht $L$ für *links*, $R$ für *rechts* und $N$ für *neutral*, also Stehenbleiben).

**Definition.** Eine *Konfiguration* einer Turingmaschine ist ein Wort $k \in \Gamma^* Z \Gamma^*$.

Inhaltlich bedeutet eine Konfiguration eine „Momentaufnahme“ der TM. Hierbei wird $k = \alpha z \beta$ so interpretiert, dass $\alpha\beta$ der nicht-leere, bzw. schon besuchte Teil des Bandes ist. $z$ ist der Zustand, in dem sich die Maschine gerade befindet, und der Schreib-Lesekopf steht auf dem ersten Zeichen von $\beta$.

Gestartet werden Turingmaschinen dadurch, dass die Eingabe $x \in \Sigma^*$ schon auf dem Band steht und der Schreib-Lesekopf auf dem ersten Zeichen von $x$. Dies wird dargestellt durch die *Startkonfiguration* $z_0 x$.

**Definition.** Wir definieren auf der Menge der Konfigurationen einer gegebenen Turingmaschine eine zweistellige Relation $\vdash$.

Es gilt:

$$a_1 \dots a_m z b_1 \dots b_n \vdash \begin{cases} a_1 \dots a_m z' c b_2 \dots b_n, \\ \quad \delta(z, b_1) = (z', c, N),\ m \geq 0,\ n \geq 1 \\ a_1 \dots a_m c z' b_2 \dots b_n, \\ \quad \delta(z, b_1) = (z', c, R),\ m \geq 0,\ n \geq 2 \\ a_1 \dots a_{m-1} z' a_m c b_2 \dots b_n, \\ \quad \delta(z, b_1) = (z', c, L),\ m \geq 1,\ n \geq 1 \end{cases}$$

Zwei Sonderfälle müssen separat definiert werden: Wenn $n = 1$ und die Maschine nach rechts läuft, so trifft sie auf ein Blank:

$$a_1 \dots a_m z b_1 \vdash a_1 \dots a_m c z' \square \text{ falls } \delta(z, b_1) = (z', c, R)$$

Wenn $m = 0$ und die Maschine nach links läuft, so trifft sie gleichfalls auf ein Blank:

$$z b_1 \dots b_n \vdash z' \square c b_2 \dots b_n \text{ falls } \delta(z, b_1) = (z', c, L)$$

Diese Definition ist so gestaltet, dass Konfigurationsbeschreibungen bei Bedarf verlängert werden, wenn die Maschine links oder rechts ein neues, bisher noch nicht besuchtes, Zeichen liest. (Dieses Zeichen kann dann nur ein Blank sein).

**Beispiel:** Gegeben sei folgende Turingmaschine, die eine Eingabe $x \in \{0,1\}^*$ als Binärzahl interpretiert und 1 hinzuaddiert:

$$M = (\{z_0, z_1, z_2, z_e\}, \{0,1\}, \{0,1,\square\}, \delta, z_0, \square, \{z_e\})$$

wobei

$$\begin{aligned}
\delta(z_0, 0) &= (z_0, 0, R) \\
\delta(z_0, 1) &= (z_0, 1, R) \\
\delta(z_0, \square) &= (z_1, \square, L) \\
\delta(z_1, 0) &= (z_2, 1, L) \\
\delta(z_1, 1) &= (z_1, 0, L) \\
\delta(z_1, \square) &= (z_e, 1, N) \\
\delta(z_2, 0) &= (z_2, 0, L) \\
\delta(z_2, 1) &= (z_2, 1, L) \\
\delta(z_2, \square) &= (z_e, \square, R)
\end{aligned}$$

Wenn diese Maschine mit der Eingabe 101 gestartet wird, so stoppt sie schließlich mit 110 auf dem Band, wobei sich der Schreib-Lesekopf wieder auf das erste nicht-leere Zeichen zurückbewegt hat.

$$z_0101 \vdash 1z_001 \vdash 10z_01 \vdash 101z_0\square \vdash 10z_11\square$$

$$\vdash 1z_100\square \vdash z_2110\square \vdash z_2\square110\square \vdash \square z_e110\square$$

Beim 3. und 7. Konfigurationsübergang trat der Sonderfall in der Definition von $\vdash$ auf.

> **Definition.** Die von einer Turingmaschine $M$ *akzeptierte Sprache* ist wie folgt definiert:
>
> $$T(M) = \{x \in \Sigma^* \mid z_0x \vdash^* \alpha z\beta;\ \alpha, \beta \in \Gamma^*;\ z \in E\}$$

Wir wollen im Folgenden noch spezielle Turingmaschinen betrachten, die den Teil des Bandes, auf dem die Eingabe steht, niemals verlassen. Diese nennen wir *linear beschränkte Turingmaschinen* (engl.: linear bounded automaton, kurz: LBA). Für eine solche Maschine ist es jedoch sinnvoll und notwendig, dass sie es erkennen kann, wenn sie sich auf einem Randfeld befindet. Das „Erkennen“ des linken Randfeldes ist kein Problem, da gerade auf diesem Feld am Anfang der Schreib-Lesekopf steht. Somit kann sich die

Maschine im ersten Rechenschritt dieses Feld „markieren“, um später nicht über diesen linken Rand hinauszulaufen.

Der rechte Rand allerdings, also das letzte Zeichen der Eingabe, wird bei der folgenden Definition jedoch schon in der Startkonfiguration besonders markiert. Hierzu verdoppeln wir das Eingabealphabet $\Sigma$ zu $\Sigma' = \Sigma \cup \{\hat{a} \mid a \in \Sigma\}$. Die „eigentliche“ Eingabe $a_1 a_2 \dots a_{n-1} a_n$ wird auf dem Band repräsentiert durch $a_1 a_2 \dots a_{n-1} \hat{a_n}$.

**Definition.** Eine nichtdeterministische Turingmaschine heißt *linear beschränkt*, wenn für alle $a_1 a_2 \dots a_{n-1} a_n \in \Sigma^+$ und alle Konfigurationen $\alpha z \beta$ mit $z_0 a_1 a_2 \dots a_{n-1} \hat{a_n} \vdash^* \alpha z \beta$ gilt: $|\alpha\beta| = n$.

Die von einer linear beschränkten Turingmaschine $M$ *akzeptierte Sprache* ist wie folgt definiert:

$$\begin{aligned} T(M) \;=\; & \{a_1 a_2 \dots a_{n-1} a_n \in \Sigma^* \mid z_0 a_1 a_2 \dots a_{n-1} \hat{a_n} \vdash^* \alpha z \beta, \\ & \alpha, \beta \in \Gamma^*;\ z \in E\} \end{aligned}$$

**Satz (Kuroda).**
DIE VON LINEAR BESCHRÄNKTEN, NICHTDETERMINISTISCHEN TURINGMASCHINEN (LBAS) AKZEPTIERBAREN SPRACHEN SIND GENAU DIE KONTEXTSENSITIVEN (TYP 1) SPRACHEN.

**Beweis:** ($\Leftarrow$) Sei $A$ eine Typ 1 Sprache, also $A = L(G), G = (V, \Sigma, P, S)$.

Wir beschreiben informal eine TM $M$, die $A$ akzeptiert: Bei Eingabe von $x = a_1 \dots a_n$ wählt $M$ zunächst nichtdeterministisch eine Produktion $u \to v \in P$ aus. Dann sucht $M$ ein beliebiges Vorkommen von $v$ auf dem Band auf. Falls ein solches gefunden werden kann, so ersetzt $M$ dieses Teilwort durch $u$. (Falls $u$ kürzer ist als $v$, so werden alle Bandsymbole rechts von $u$ entsprechend nach links verschoben). Falls der nicht-leere Teil des Bandes nur noch die Startvariable $S$ enthält, so stoppt $M$ in einem Endzustand. Ansonsten werden diese nichtdeterministischen Ersetzungsvorgänge wiederholt.

Nun gilt:

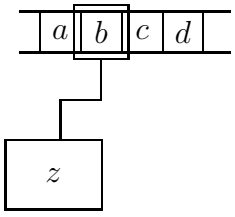
$$\begin{aligned} x \in L(G) \quad & \mathbf{gdw} \quad \text{es gibt eine Ableitung } S \Rightarrow \dots \Rightarrow x \\ & \mathbf{gdw} \quad \text{es gibt eine Rechnung von } M, \text{ die diese Ableitung} \\ & \qquad\quad \text{in umgekehrter Richtung simuliert} \\ & \mathbf{gdw} \quad x \in T(M) \end{aligned}$$

Da für die Regeln $u \to v \in P$ gilt $|u| \leq |v|$, ist $M$ linear beschränkt.

($\Rightarrow$) Sei umgekehrt $A = T(M)$ für eine linear beschränkte Turingmaschine $M$. Im folgenden beschreiben wir eine kontextsensitive Grammatik, die auf Wörtern, die Konfigurationen von $M$ darstellen, operiert. Hierzu müssen wir Konfigurationen so beschreiben,

dass sie aus nicht mehr Zeichen bestehen als die Eingabe, also der nicht-leere Teil des Bandes zu Beginn. Wir wählen zu diesem Zweck das Alphabet $\Delta = \Gamma \cup (Z \times \Gamma)$.

Die Konfiguration




wird beispielweise dargestellt durch $a(z,b)cd$. Dieses Wort hat nur die Länge $4 = |abcd|$ (bezogen auf das zugrunde liegende, erweiterte Alphabet $\Delta$).

$\delta$-Übergänge von $M$, etwa

$$\delta(z,a) \ni (z',b,L)$$

können durch (kontextsensitive) Produktionen beschrieben werden:

$$c\,(z,a) \to (z',c)\,b \quad \text{für alle } c \in \Gamma.$$

Die so entstehende Produktionenmenge nennen wir $P'$. Falls also $k \vdash^* k'$ unter $M$ gilt, so gilt mittels $P'$: $\tilde{k} \Rightarrow^* \tilde{k'}$ (und umgekehrt), wobei $\tilde{k}$ die oben angegebene Darstellung der Konfiguration $k$ ist.

Die gesuchte kontextsensitive Grammatik $G = (V, \Sigma, P, S)$ sieht nun wie folgt aus:

$$\begin{aligned}
V &= \{S, A\} \cup (\Delta \times \Sigma) \\
P &= \{S \to A(\hat{a}, a) \mid a \in \Sigma\} && (1)\\
&\quad \cup\{A \to A(a,a) \mid a \in \Sigma\} && (2)\\
&\quad \cup\{A \to ((z_0,a),a) \mid a \in \Sigma\} && (3)\\
&\quad \cup\{(\alpha_1,a)(\alpha_2,b) \to (\beta_1,a)(\beta_2,b) \mid \alpha_1\alpha_2 \to \beta_1\beta_2 \in P', \\
&\qquad a,b \in \Sigma\} && (4)\\
&\quad \cup\{((z,a),b) \to b \mid z \in E,\ a \in \Gamma,\ b \in \Sigma\} && (5)\\
&\quad \cup\{(a,b) \to b \mid a \in \Gamma,\ b \in \Sigma\} && (6)
\end{aligned}$$

Die Idee hierbei ist die folgende: Zunächst sind mittels (1),(2),(3) Ableitungen möglich der Form

$$S \Rightarrow^* ((z_0,a_1),a_1)(a_2,a_2)\ldots(a_{n-1},a_{n-1})(\hat{a_n},a_n)$$

Die ersten Komponenten stellen eine Startkonfiguration (repräsentiert über $\Delta$) dar, die zweiten Komponenten das zugehörige Eingabewort.

Nun wird auf den ersten Komponenten mittels $P'$ (Regelart (4)) eine Rechnung von $M$ simuliert, bis ein Endzustand erreicht wird:

$$\ldots \Rightarrow^* (\gamma_1,a_1)\ldots(\gamma_{k-1},a_{k-1})((z,\gamma_k),a_k)(\gamma_{k+1},a_{k+1})\ldots(\gamma_n,a_n)$$
$$\text{mit } z \in E, \gamma_i \in \Gamma, a_i \in \Sigma$$

Danach können mittels (5),(6) alle ersten Komponenten weggelöscht werden. Es bleibt $a_1 \dots a_n$ übrig:

$$\dots \Rightarrow^* a_1 \dots a_n$$

Man prüft leicht nach, dass alle Regeln vom Typ 1 sind. □

Unter Weglassen der linearen Beschränktheit von $M$ bzw. der Typ 1-Bedingung von $G$ erhalten wir aus obigem Beweis sofort einen Beweis für den folgenden

**Satz.**
DIE DURCH ALLGEMEINE TURINGMASCHINEN AKZEPTIERBAREN SPRACHEN SIND GENAU DIE TYP 0-SPRACHEN.

*Bemerkungen:* Der Berechnungsbaum einer nichtdeterministischen TM kann von einer deterministischen Turingmaschine systematisch durchsucht werden (nach einer Konfiguration mit Endzustand) – allerdings ist nicht klar, ob die simulierende Maschine hernach immer noch linear beschränkt ist, wenn es die ursprüngliche war.

Das bedeutet: im allgemeinen (Typ 0) Fall spielt es keine Rolle, ob wir von nichtdeterministischen oder von deterministischen Turingmaschinen reden, denn nichtdeterministische Turingmaschinen können durch deterministische simuliert werden. In der Typ 1-Situation allerdings war für den Äquivalenzbeweis eine *nichtdeterministische* Turingmaschine notwendig. Ob man in diesem Fall auch mit einer deterministischen auskommen kann, ist ein bis heute ungelöstes Problem, das sog. LBA-Problem. Auf einen kurzen, formelhaften Nenner gebracht, lautet also die Frage, ob $LBA = DLBA$.

Es gilt jedoch der folgende (erstaunliche?) Satz.

**Satz (Immerman, Szelepcsényi).**
DIE KLASSE DER KONTEXTSENSITIVEN (ALSO TYP 1) SPRACHEN IST UNTER KOMPLEMENTBILDUNG ABGESCHLOSSEN.

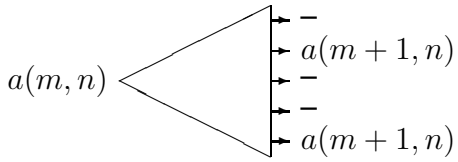
**Beweis:** Sei $L = L(G) \subseteq \Sigma^*$ eine Typ 1 Sprache. Wir geben einen LBA $M$ an, der das Komplement von $L$ akzeptiert. Sei $V$ die Variablenmenge von $G$. Bei Eingabe $x$, $|x| = n$, berechnet $M$ zunächst die exakte Anzahl $a \in I\!N$ der von der Startvariablen $S$ aus erzeugbaren Satzformen mit bis zu $n$ Zeichen. (In der Notation von Seite 62 ist also $a = |T_m^n|$, wobei $m$ so groß ist, dass $T_m^n = T_{m+1}^n$). Wir beobachten, dass die Binärdarstellung der Zahl $a$ nur linear in $n$ viel Platz auf dem Band einnimmt. Sodann arbeitet $M$ wie folgt: In einer Schleife wird jedes mögliche Wort der Länge $\leq n$, bis auf $x$, über dem Alphabet $V \cup \Sigma$ systematisch aufgezählt. Für jedes dieser Wörter $w$ wird in nichtdeterministischer Art und Weise geprüft, ob $S \Rightarrow_G^* w$. Hierbei besteht aber auch die nichtdeterministische Alternative, mit „Mißerfolg“ die Schleife fortzusetzen. Im „Erfolgsfall“ wird jedoch ein Zähler hochgezählt. Nur wenn dieser Zähler nach Ablauf der Schleife den Wert $a$ erreicht hat, so akzeptiert die Maschine $M$. Dies bedeutet nämlich, dass $M$ in der Lage war,

sämtliche Satzformen der Länge $\leq n$ zu generieren (und als solche zu erkennen), wobei hierbei $x$ nicht vorkam. Also ist dies genau dann der Fall, wenn $x \notin L$. Man beachte, dass die Schleife mit linearem Platz programmiert werden kann.

Es verbleibt noch zu zeigen, wie die Zahl $a$ berechnet werden kann. Sei $a(m,n)$ die Anzahl der in $\leq m$ Schritten von der Startvariablen $S$ erzeugbaren Satzformen der Länge $\leq n$ (also $a(m,n) = |T_m^n|$). Wir geben einen nichtdeterministischen Algorithmus an, der unter der Voraussetzung, dass $a(m,n)$ bekannt ist, die Zahl $a(m+1,n)$ berechnet. Insgesamt startet der Algorithmus dann mit $a(0,n) = |\{S\}| = 1$ und iteriert das Verfahren solange, bis $a(m,n) = a(m+1,n)$ gilt (vgl. Seite 62). Dies ist dann die gesuchte Zahl $a$. Diese Vorgehensweise wird manchmal *induktives Zählen* genannt.

Bei Eingabe von $a(m,n)$ berechnet $M$ die Zahl $b = a(m+1,n)$ wie folgt. Zunächst wird $b$ mit 0 initialisiert. Analog dem oben beschriebenen Verfahren werden dieses Mal in zwei ineinander verschachtelten Schleifen alle Paare von Wörtern $w, w'$ bis zur Länge $n$ über dem Alphabet $V \cup \Sigma$ generiert. Dabei wird $w'$ in der äußeren und $w$ in der inneren Schleife generiert. Vor jedem inneren Schleifendurchlauf wird ein weiterer Zähler $z$ mit 0 initialisiert und im Inneren der $w$-Schleife wird nichtdeterministisch überprüft, ob es eine Ableitung von $S$ nach $w$ in $\leq a(m,n)$ Schritten gibt. Im Erfolgsfall wird der Zähler $z$ hochgezählt und ferner überprüft, ob $w = w'$ oder $w \Rightarrow_G w'$ gilt. Sollte dies der Fall sein, so wird der Zähler $b$ hochgezählt. Nach jeder Beendigung der inneren Schleife wird die Rechnung nur dann fortgesetzt, wenn der Zähler $z$ den Wert $a(m,n)$ erreicht hat, ansonsten wird verworfen. Nach Beendigung der äußeren Schleife hat der Zähler $b$ dann den Wert $a(m+1,n)$.

Die folgende Skizze zeigt schematisch den nichtdeterministischen Rechenablauf, der von $a(m,n)$ zu $a(m+1,n)$ führt. Die nichtdeterministischen Rechnungen enden entweder verwerfend (–) oder mit dem korrekten Wert $a(m+1,n)$.




Man kann den gesamten Rechenablauf so organisieren, dass nicht mehr als $n$ Bandfelder verwendet werden. Daher ist $M$ ein LBA, und $M$ akzeptiert die Sprache $\overline{L}$. □

## 3.5 Tabellarischer Überblick

Wir stellen nun in Tabellenform die wichtigsten Resultate zusammen. Um die Tabellen jedoch vollständig zu machen, sind an der einen oder anderen Stelle auch Ergebnisse eingefügt, die im Text nicht behandelt wurden.

**Beschreibungsmittel.** Mit welchen Mitteln der Beschreibung kann welcher Sprachtyp dargestellt werden? Meist haben wir zumindest eine Grammatikart und einen äquivalenten Automaten kennengelernt.

| | |
|---|---|
| Typ 3 | reguläre Grammatik<br>DFA<br>NFA<br>regulärer Ausdruck |
| Det. kf. | $LR(k)$-Grammatik<br>deterministischer Kellerautomat (DPDA) |
| Typ 2 | kontextfreie Grammatik<br>Kellerautomat (PDA) |
| Typ 1 | kontextsensitive Grammatik<br>linear beschränkter Automat (LBA) |
| Typ 0 | Typ 0 - Grammatik<br>Turingmaschine (TM) |

**Determinismus und Nichtdeterminismus.** Wir stellen zusammen, inwieweit bei den verschiedenen Automatenmodellen die deterministische und die nichtdeterministische Version äquivalent sind.

| Nichtdet. Automat | Determ. Automat | äquivalent? |
|---|---|---|
| NFA | DFA | ja |
| PDA | DPDA | nein |
| LBA | DLBA | ? |
| TM | DTM | ja |

Die Frage, ob sich nichtdeterministische LBAs äquivalent in deterministische umformen lassen, ist ungelöst. Diese Frage ist als *LBA-Problem* bekannt.

**Abschlusseigenschaften.** Die betrachteten Sprachklassen sind (sind nicht) abgeschlossen unter den folgenden Operationen.

| | Schnitt | Vereinigung | Komplement | Produkt | Stern |
|---|---|---|---|---|---|
| Typ 3 | ja | ja | ja | ja | ja |
| Det. kf. | nein | nein | ja | nein | nein |
| Typ 2 | nein | ja | nein | ja | ja |
| Typ 1 | ja | ja | ja | ja | ja |
| Typ 0 | ja | ja | nein | ja | ja |

**Entscheidbarkeit.** Die folgenden Fragestellungen sind (sind nicht) entscheidbar. Die Unentscheidbarkeitsergebnisse können erst in der Vorlesung Berechenbarkeit bewiesen werden.

Hierbei bedeutet:

*Wortproblem*: Gegeben ein Wort $x$ (und ein Automat $M$ oder eine Grammatik $G$). Liegt $x$ in $T(M)$ bzw. in $L(G)$?

*Leerheitsproblem*: Gegeben ein Automat $M$ oder eine Grammatik $G$. Ist $T(M)$ bzw. $L(G)$ die leere Menge?

*Endlichkeitsproblem*: Gegeben ein Automat $M$ oder eine Grammatik $G$. Ist $T(M)$ bzw. $L(G)$ endlich?

*Äquivalenzproblem*: Gegeben zwei Automaten $M_1, M_2$ oder zwei Grammatik $G_1, G_2$. Gilt $T(M_1) = T(M_2)$ bzw. $L(G_1) = L(G_2)$?

*Schnittproblem*: Gegeben zwei Automaten $M_1, M_2$ oder zwei Grammatik $G_1, G_2$. Gilt $T(M_1) \cap T(M_2) = \emptyset$ bzw. $L(G_1) \cap L(G_2) = \emptyset$?

| | Wort-problem | Endlichkeits/ Leerheits-problem | Äquivalenz-problem | Schnitt-problem |
|---|---|---|---|---|
| Typ 3 | ja | ja | ja | ja |
| Det. kf. | ja | ja | ja | nein |
| Typ 2 | ja | ja | nein | nein |
| Typ 1 | ja | nein | nein | nein |
| Typ 0 | nein | nein | nein | nein |

Bis 1997 war ungelöst, ob das Äquivalenzproblem bei deterministisch kontextfreien Sprachen entscheidbar ist. Mit einem sehr aufwändigen Beweis wurde die Entscheidbarkeit von Senizergues 1997 gezeigt.

# Literatur


U. Schöning, H.A. Kestler: Mathe-Toolbox. Lehmanns, 2012.

P. Clote, E. Kranakis: Boolean Function Theory. Springer, 2002.

I. Wegener: The Complexity of Boolean Functions. Wiley-Teubner, 1987.

I. Wegener: Effiziente Algorithmen für grundlegende Funktionen. Teubner, 1989.

S. Jukna: Boolean Function Complexity. Springer, 2012.

M.L. Minsky, S.A. Papert: Perceptrons. MIT Press, 1988.

I. Parberry: Circuit Complexity and Neural Networks. MIT Press, 1994.

R. Rojas: Theorie der neuronalen Netze. Springer, 1993.

B. Lenze: Einführung in die Mathematik neuronaler Netze, Logos Verlag, 2003.

J. Hertz, A. Krogh, R.G. Palmer: Introduction to the Theory of Neural Computation. Addison-Wesley, 1991.

W. Lütkebohmert: Codierungstheorie. Vieweg, 2003.

M. Bossert: Kanalcodierung. Teubner, 1998.

R.H. Schulz: Codierungstheorie. Vieweg, 1991.

G. Schmidt, T. Ströhlein: Relationen und Graphen. Springer, 1989.

A. Steger: Diskrete Strukturen. Springer, 2001.

T. Ihringer, Diskrete Mathematik. Teubner, 1994.

M. Aigner, Diskrete Mathematik. Vieweg, 1993.

U. Knauer: Diskrete Strukturen kurz gefasst. Spektrum, 2001.

R. Diestel: Graphentheorie. Springer, 1996.

U. Schöning: Theoretische Informatik – kurz gefasst. Spektrum, 2008.

J.E. Hopcroft, R. Motwanji, J.D. Ullman: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 2001.

M. Sipser: Introduction to the Theory of Computation. PWS Publishing Company, 1997.

# Index