Informatik IV

Theoretische Grundlagen der Informatik

Prof. Dr. Alfred Widiger

Gehalten im Sommersemester 2006
an der **Universität Rostock**

Informatik IV

Theoretische Grundlagen der Informatik

Prof. Dr. Alfred Widiger

Gehalten im Sommersemester 2006
an der **Universität Rostock**

Zusammenfassung

Gegenstand der Vorlesung

- Modelle (von Rechnern)
- Vergleich der Modelle (bzgl. Leistungsfähigkeit)
- Gibt es ein „*bestes*“ Modell? (mit dem man alles leisten kann)
- Wie kann man Algorithmen allgemein bzgl. Komplexibilität klassifizieren?
- Gibt es Probleme, die zwar prinzipiell lösbar sind, aber praktisch unlösbar?

Inhaltsverzeichnis

Kapitel 1

Automaten und formale Sprachen

1.1 Endliche Automaten mit Ausgabe

1.1.1 Grundlagen

Definition (*Automat*)**.** System, das in der Lage ist, sein Verhalten (ohne unmittelbares Eingreifen des Menschen) selbst zu steuern.

Ein Automat:

- hat innere Zustände (endlich viele)
- erhält einen Input $\searrow$
- gibt einen Output $\rightarrow$ Kommunikation mit Umwelt

Seine Beschreibung erfordert:

- Welche inneren Zustände kann er annehmen?
- Welche Eingabe kann er aufnehmen?
- Welche Ausgabe kann er ausgeben?
- Welchen Zustand nimmt er an, wenn in einem bestimmten Zustand eine bestimmte Eingabe erfolgt?
- Welche Ausgabe macht er, wenn in einem bestimmten Zustand eine bestimmte Eingabe erfolgt?

Beispiel (Mausefalle M)**.** Die Mausefalle habe zwei Zustände

- z_1: Falle gespannt
- z_2: Falle nicht gespannt,

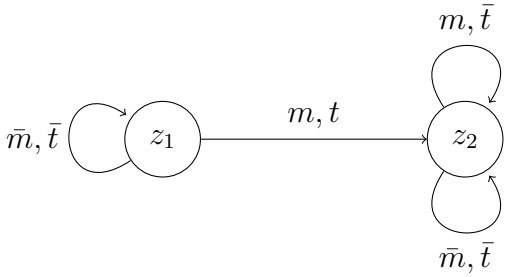
welche die *Zustandsmenge*

$$Z = \{z_1, z_2\}$$

bilden.
Menge der Eingabezeichen: $X = \{m, \bar{m}\}$ mit m: Maus kommt, $\bar{m}$: Maus kommt nicht
Menge der Ausgabezeichen: $Y = \{t, \bar{t}\}$ mit t: Maus ist tot, $\bar{t}$: Maus ist nicht tot

Verhalten des Automaten:

$Z\backslash X$	m	$\bar{m}$
z_1	z_2, t	$z_1, \bar{t}$
z_2	$z_2, \bar{t}$	$z_2, \bar{t}$



Beispiel. Erweiterung des Modells durch Berücksichtigung von Speck (Automat M').

$Z = \{z'_1, z'_2, z'_3, z'_4\}$ mit

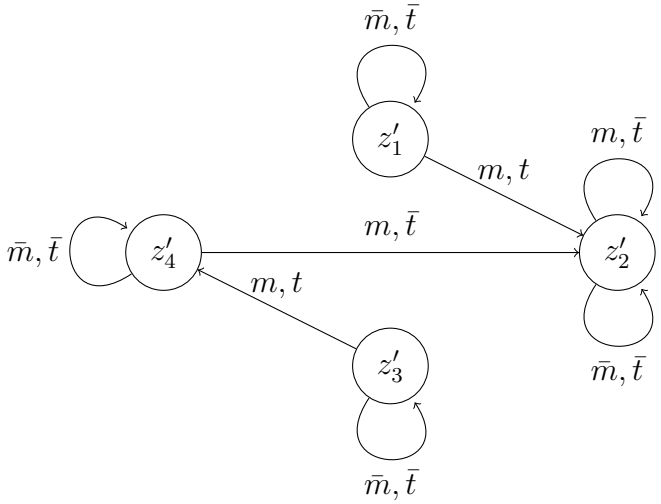
z'_1**:** ohne Speck, Falle gespannt

z'_2**:** ohne Speck, Falle nicht gespannt

z'_3**:** mit Speck, Falle gespannt

z'_4**:** mit Speck, Falle nicht gespannt

$Z\backslash X$	m	$\bar{m}$
z'_1	z'_2, t	$z'_1, \bar{t}$
z'_2	$z'_2, \bar{t}$	$z'_2, \bar{t}$
z'_3	z'_4, t	$z'_3, \bar{t}$
z'_4	$z'_2, \bar{t}$	$z'_4, \bar{t}$



Definition (MEALY-*Automat*)**.** Ein MEALY-Automat ist ein 5-Tupel $A = (X, Y, Z, f, g)$ mit nicht leeren endlichen Mengen X, Y, Z und Abbildungen

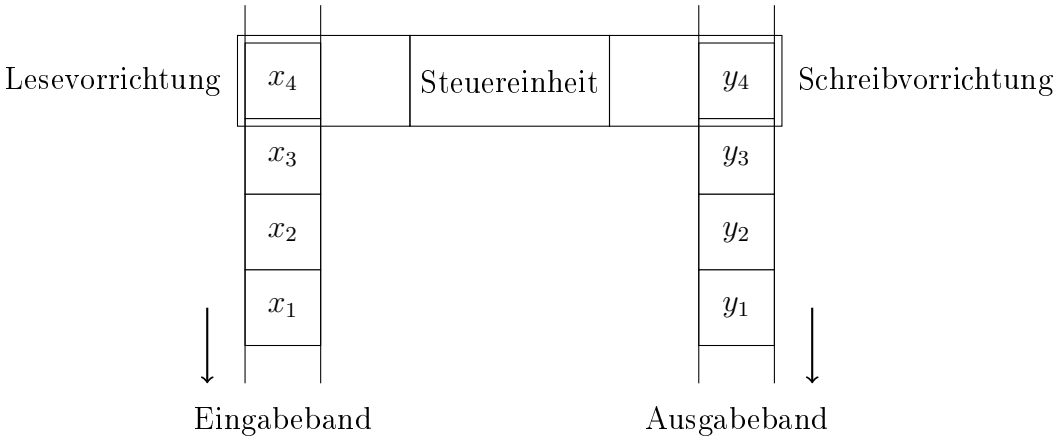
$$f : Z \times X \to Z$$
$$g : Z \times X \to Y$$

Hierbei bedeuten:

- X: Eingabealphabet
- Y: Ausgabealphabet

- Z: Zustandsmenge
- f: Überführungsfunktion
- g: Ausgabefunktion

Modell



Arbeitsweise

Anfangszustand z setzen
LV auf 1. Zeichen der Eingabe setzen
`while`: Feld unter Lesekopf nicht leer `do` - Zeichen X und Lesekopf lesen - Auf Ausgabeband Zeichen $y = g(z, x)$ schreiben - Zustand $z := f(z, x)$ - Ausgabeband um ein Feld weiterrücken - Eingabeband um ein Feld weiterrücken

Darstellung in Tabellenform

$$X = \{x_1, \ldots, x_n\}$$
$$Y = \{y_1, \ldots, y_n\}$$
$$Z = \{z_1, \ldots, z_n\}$$

Überführungsfunktion

$Z \backslash X$	x_1	x_2	$\ldots$	x_n
z_1	$f(z_1, x_1)$	$f(z_1, x_2)$	$\ldots$	$f(z_1, x_n)$
$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\vdots$
z_n	$f(z_n, x_1)$	$f(z_n, x_2)$	$\ldots$	$f(z_n, x_n)$

Ausgabefunktion

$Z \backslash X$	x_1	x_2	$\ldots$	x_n
z_1	$g(z_1, x_1)$	$g(z_1, x_2)$	$\ldots$	$g(z_1, x_n)$
$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\vdots$
z_n	$g(z_n, x_1)$	$g(z_n, x_2)$	$\ldots$	$g(z_n, x_n)$

Darstellung mittels gerichteter Graphen

Definition (*gerichteter Graph*)**.** Das Paar $G = (K, R)$ heißt gerichteter Graph, wenn K eine nichtleere Menge und $R \subseteq K \times K$, d.h. R ist 2-stellige Relation.

- K Knotenmenge
- R Kantenmenge
- Für $(k, k') \in R$ heißt k Anfangsknoten und k' Endknoten

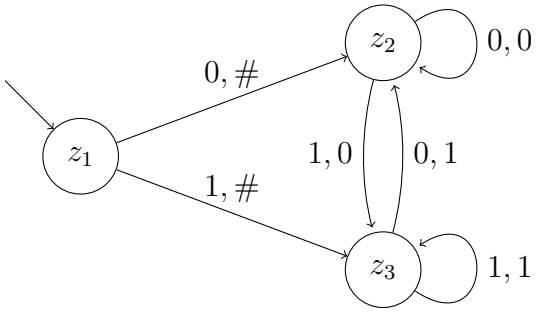
Dem MEALY-Automaten $A = (X, Y, Z, f, g)$ wird der gerichtete Graph $G_A = (Z, R)$ mit

$$R = \{(z, z') : z, z' \in Z \wedge \exists x \in X : f(z, x) = z'\}$$

zugeordnet. Dabei wird eine Kante (z, z') bei $f(z, x) = z'$ mit x und $g(z, x)$ beschriftet. Dieser Graph heißt Zustandsgraph von A.

Beispiel. Es soll ein MEALY-Automat entworfen werden, der ein Eingabewort um ein Zeichen versetzt reproduziert. Die Ausgabe beginnt immer mit #, das letzte Zeichen des Inputwortes geht verloren.
Es sei $X = \{0, 1\}$, $Y = \{0, 1, \#\}$ z.B.: Eingabe: 011011, Ausgabe: #01101
z_2: letztes Zeichen 0, z_3: letztes Zeichen 1

$Z \backslash X$	0	1
z_1	$z_2, \#$	$z_3, \#$
z_2	$z_2, 0$	$z_3, 0$
z_3	$z_2, 1$	$z_3, 1$



1.1.2 Äquivalenz und Reduktion

Alphabet M**:** endliche nichtleere Menge (von Zeichen)

Wörter über M**:** endliche Folge von Zeichen aus M (Strings)

M^***:** Menge aller Wörter über M

Konkatenation von Wörtern über M**:** (Operation auf M^*) Diese Operation ist assoziativ

$$[(pq)r = p(qr) \quad \forall p, q, r \in M^*]$$

ε**:** leeres Wort über M. Ist bezüglich Konkatenation neutrales Element

$$(\varepsilon p = p\varepsilon = p \quad \forall p \in M^*)$$

Definition (*Monoid, Halbgruppe mit 1*)**.** Menge mit assoziativer Verknüpfung und neutralem Element. Wenn $w \in M^*$ sei $|w|$ die Länge von w (d.h. die Anzahl von Zeichen in w) z.B.

$$|\varepsilon| = 0,\ |m_1 m_2| = 2 \quad (m_1, m_2 \in M)$$

Für die Beurteilung des Verhaltens eines MEALY-Automaten ist entscheidend, welche Ausgabezeichenfolge $y_{t_1} \dots y_{t_r} \in Y^*$ er bei Eingabe einer Zeichenfolge $x_{t_1} \dots x_{t_r} \in X^*$ produziert.

Definition. Sei $A = (X, Y, Z, f, g)$ ein MEALY-Automat.

$$f^* : Z \times X^* \to Z \qquad\qquad g^* : Z \times X^* \to Y^*$$

seien definiert als $f^*(z, \varepsilon) = z\ ,\ \ g^*(z, \varepsilon) = \varepsilon$

$$f^*(z, px) = f(f^*(z, p), x) \qquad\qquad g^*(z, px) = g^*(z, p) g(f^*(z, p), x)$$

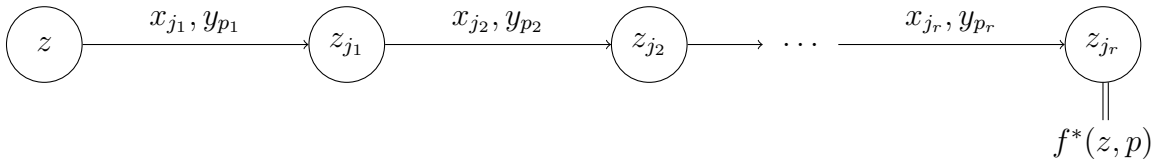
$$\forall z \in Z,\ x \in X,\ p \in X^*$$

f^***:** erweiterte Überführungsfunktion

g^***:** erweiterte Ergebnisfunktion

Die Abbildung $g_z^* : X^* \to Y^*$ mit $g_z^*(p) = g^*(z, p)$ heißt *Leistung* von z.

Bemerkung. Sei $p = x_{r_1} x_{r_2} \dots x_{r_n} \in X^*,\ [x = \{x_1, \dots, x_n\}]$



$$y_{i_1} y_{i_2} \dots y_{i_n} = g^*(z, p) = g_z^*(p)$$
$$\left.\begin{array}{rcl} g^*(z, x) & = & g(z, x) \\ f^*(z, x) & = & f(z, x) \end{array}\right\} x \in X$$

Lemma 1. $\forall p, q \in X^*$ gilt:

- $f^*(z, pq) = f^*(f^*(z, p), q)$
- $g^*(z, pq) = g^*(z, p) g^*(f^*(z, p), q)$

Das Lemma ist anschaulich klar.

Beweis. (zu Lemma (1) (formal))
Für $q = \varepsilon$ richtig.
Für $q \neq \varepsilon$ mit vollständiger Induktion nach $|q|$.
Induktionsanfang $|q| = 1$, d.h. $q = x \in X$

$$\begin{array}{rcl} f^*(z, px) & \overset{\text{Def}}{=} & f(f^*(z, p), x) \\ & = & f^*(f^*(z, p), x) \\ g^*(z, px) & \overset{\text{Def}}{=} & g^*(z, p) g(f^*(z, p), x) \\ & = & g^*(z, p) g^*(f^*(z, p), x) \end{array}$$

Angenommene Behauptung gilt für q mit $|q| \leq n$.

Sei $q = q_1 x, x \in X, |q| = n+1$.

$$\begin{aligned} f^*(z, pq) &= f^*(z, (pq_1)x) \\ &= f^*(f^*(z, pq_1), x) \\ &\overset{\text{IV}}{=} f^*(f^*(f^*(z,p), q_1), x) \\ &= f^*(f^*(z,p), q_1 x) \\ &= f^*(f^*(z,p), q) \\ g^*(z, pq) &= g^*(z, pq_1 x) \\ &= \dots \end{aligned}$$

□

Beispiel. Für das letzte Bsp. in Abschnitt (1.1.1) wird offenbar

$$\begin{aligned} g^*_{z_1}(x_{i_1} \dots x_{i_r}) &= \#x_{i_1} \dots x_{i_{r-1}} \\ g^*_{z_2}(x_{i_1} \dots x_{i_r}) &= 0x_{i_1} \dots x_{i_{r-1}} \\ g^*_{z_3}(x_{i_1} \dots x_{i_r}) &= 1x_{i_1} \dots x_{i_{r-1}} \end{aligned}$$

Definition (*Äquivalente Zustände*)**.** Zwei Zustände $z_1, z_2 \in Z$ des MEALY-Automaten $A = (X, Y, Z, f, g)$ heißen *äquivalent* - $z_1 \sim z_2$, wenn sie die gleiche Leistung haben, d.h. $g^*_{z_1} = g^*_{z_2}$ ist.

Definition. Sind $A_1 = (X, Y, Z_1, f_1, g_1)$ und $A_2 = (X, Y, Z_2, f_2, g_2)$ zwei MEALY-Automaten (mit gleichem Eingabealphabet X und gleichem Ausgabealphabet Y), so heißen:

- die Zustände $z_1 \in Z_1$ und $z_2 \in Z_2$ äquivalent, wenn sie die gleiche Leistung haben, d.h. wenn $g^*_{1,z_1} = g^*_{2,z_2}$ ist.
- A_1 und A_2 äquivalent, wenn die Menge ihrer Leistungen gleich sind, d.h.

$$\left\{g^*_{1,z_1} : z_1 \in Z_1\right\} = \left\{g^*_{2,z_2} : z_2 \in Z_2\right\}$$

  gilt.

Offenbar gilt:

Satz 1.1. Die MEALY-Automaten $A_1 = (X, Y, Z_1, f_1, g_1)$ und $A_2 = (X, Y, Z_2, f_2, g_2)$ sind genau dann äquivalent, wenn es zu jedem Zustand $z_1 \in Z_1$ einen Zustand $z_2 \in Z_2$ gibt, mit $z_1 \sim z_2$ und wenn es zu jedem Zustand $z'_2 \in Z_2$ einen Zustand $z'_1 \in Z_1$ mit $z'_1 \sim z'_2$ gibt.

Beispiel. Mausefalle M und M'. Sei $M_1 = M'$ und $M_2 = M$.
Offenbar ist $g^*_{2,z_2}(x_1 \dots x_s) = \underbrace{\bar{t}\,\bar{t} \dots \bar{t}}_{s}$.

Für g^*_{2,z_1} beachte man: Sobald im Eingabewort m vorkommt, geht $M_2 = M$ in z_2 über und dann werden nur noch $\bar{t}$ ausgegeben.

$$\Rightarrow g^*_{2,z_1}(\underbrace{\bar{m} \dots \bar{m}}_{l}\ m\ x_{l+2} \dots x_s) = \underbrace{\bar{t} \dots \bar{t}}_{l}\ \underbrace{t\,\bar{t} \dots \bar{t}}_{s-l}; \quad 0 \leq l \leq s$$

Für $M' = M_1$ wird

$$\begin{aligned}
g^*_{1,z'_2}(x_1 \dots x_s) &= \underbrace{\bar{t} \dots \bar{t}}_{s} \\
g^*_{1,z'_4}(x_1 \dots x_s) &= \underbrace{\bar{t} \dots \bar{t}}_{s} \\
g^*_{1,z'_1}(\underbrace{\bar{m} \dots \bar{m}}_{l}\ m\ x_{l+2} \dots x_s) &= \underbrace{\bar{t} \dots \bar{t}}_{l}\ \underbrace{t\,\bar{t} \dots \bar{t}}_{s-l}; \quad 0 \le l \le s \\
g^*_{1,z'_3}(\underbrace{\bar{m} \dots \bar{m}}_{l}\ m\ x_{l+2} \dots x_s) &= \underbrace{\bar{t} \dots \bar{t}}_{l}\ \underbrace{t\,\bar{t} \dots \bar{t}}_{s-l}; \quad 0 \le l \le s
\end{aligned}$$

Folgerung. $z_1 \not\sim z_2$; $z'_1 \sim z'_3$; $z'_2 \sim z'_4$; $z'_1 \not\sim z'_2$
$z_2 \sim z'_2$; $z_2 \sim z'_4$; $z_1 \sim z'_1$; $z_1 \sim z'_3$; $M_1 \sim M_2$

Lemma 2. $z_1 \sim z_2 \Longrightarrow f^*(z_1, p) \sim f^*(z_2, p)\ \forall p \in X^*$
(d.h. *Folgezustände* sind äquivalent)

Beweis.

$$\begin{aligned}
z_1 \sim z_2 &\Longrightarrow g^*(z_1, pq) = g^*(z_2, pq) \\
&\overset{L(1)}{\Longrightarrow} g^*(z_1, p) g^*(f^*(z_1, p), q) = g^*(z_2, p) g^*(f^*(z_2, p), q) \\
&\Longrightarrow g^*(z_1, p) = g^*(z_2, p) \\
&\Longrightarrow g^*(f^*(z_1, p), q) = g^*(f^*(z_2, p), q) \quad \forall p, q \in X^* \\
&\Longrightarrow f^*(z_1, p) \sim f^*(z_2, p)
\end{aligned}$$

□

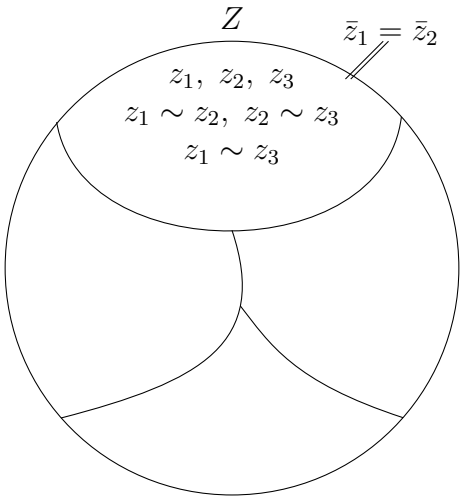
Reduktion von Automaten

Definition. Der MEALY-Automat $A = (X, Y, Z, f, g)$ heißt *reduziert*, wenn je zwei verschiedene Zustände von A nicht äquivalent sind.
Offenbar sind reduzierte Automaten wünschenswert, weil sie die gleiche Leistung mit weniger Zuständen bringen.

Frage. Gibt es zu jedem Automaten einen äquivalenten Reduzierten?

Satz 1.2. Sei $A = (X, Y, Z, f, g)$ ein MEALY-Automat. Dann existiert ein MEALY-Automat A_R, der zu A äquivalent ist. A_R ist eindeutig bestimmt.

Beweis. Die Äquivalenz (von Zuständen) ist eine Äquivalenzrelation auf Z.
(Relation ist: *reflektiv* ($z \sim z$), *symmetrisch* ($z_1 \sim z_2 \Longrightarrow z_2 \sim z_1$), *transitiv* ($z_1 \sim z_2, z_2 \sim z_3 \Longrightarrow z_1 \sim z_3$))



Z zerfällt daher in Äquivalenzklassen. Sei $\bar{Z}$ die Menge der Äquivalenzklassen und $\bar{z}$ die Klasse mit $z \in \bar{z}$. Man definiert

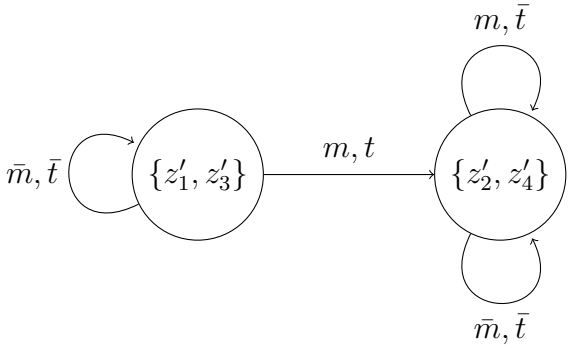
$$A_{\mathrm{R}} = (X, Y, \bar{Z}, f_{\mathrm{R}}, g_{\mathrm{R}}) \text{ mit } f_{\mathrm{R}}(\bar{z}, x) := \overline{f(z,x)} \qquad f_{\mathrm{R}} : \bar{Z} \times X \to \bar{Z}$$

$$g_{\mathrm{R}}(\bar{z}, x) := g(z, x)$$

f_{R} und g_{R} sind *wohldefiniert* (unabhängig von der Wahl des Repräsentanten), denn ist etwa $z_1 \in \bar{z}$, so gilt $z_1 \sim z$, also $g(z_1, x) = g(z, x)$. Nach Lemma (2) gilt auch $f(z_1, x) \sim f(z, x)$. Man kann zeigen, dass A_{R} reduziert ist. □

Beispiel. Für die Mausefalle mit Speck hat man

$$\overline{Z} = \{\{z_1', z_3'\}, \{z_2', z_4'\}\} \quad \text{und damit}$$



M_{R}' ist also *isomorph* zur Mausefalle M!

Problem. bei der Konstruktion von A_{R}: Wie bestimmt man die Äquivalenzklassen?

Definition (*k-Äquivalenz*)**.** Zwei Zustände z_1 und z_2 des MEALY-Automaten $A = (X, Y, Z, f, g)$ heißen k-äquivalent, wenn $\forall p \in X^*$ mit $|p| = k \quad g^*(z_1, p) = g^*(z_2, p)$

Bemerkung.

1. $z_1 \overset{k}{\sim} z_2 \Longrightarrow z_1 \overset{l}{\sim} z_2 \ \forall l \leq k$ (sofort nach Lemma (2))
2. $z_1 \sim z_2 \Longleftrightarrow z_1 \overset{k}{\sim} z_2 \ \forall k \geq 1$
3. $z_1 \overset{1}{\sim} z_2 \Longleftrightarrow g(z_1, x) = g(z_2, x) \ \ \forall x \in X$

Lemma 3. Für $k \geq 2$: $z_1 \overset{k}{\sim} z_2 \Longleftrightarrow z_1 \overset{1}{\sim} z_2$ und $f(z_1,x) \overset{k-1}{\sim} f(z_2,x)\ \forall x \in X$

Beweis.

$$\begin{aligned} z_1 \overset{k}{\sim} z_2 &\Longleftrightarrow g^*(z_1,p) = g^*(z_2,p)\ \forall p = xq \in X^*\ ,\ |p| = k \\ &\Longleftrightarrow g(z_1,x)g^*(f(z_1,x),q) = g(z_2,x)g^*(f(z_2,x),q)\ \forall x \in X\ ,\ q \in X^*\ ,\ |q| = k-1 \\ &\Longleftrightarrow g(z_1,x) = g(z_2,x)\ \forall x \in X \\ &\qquad \text{und}\ \ g^*(f(z_1,x),q) = g^*(f(z_2,x),q)\ \forall q \in X^*\ ,\ |q| = k-1 \\ &\Longleftrightarrow z_1 \overset{1}{\sim} z_2 \quad \text{und} \quad f(z_1,x) \overset{k-1}{\sim} f(z_2,x)\ \forall x \in X \end{aligned}$$

□

Die k-Äquivalenz ist eine Äquivalenzrelation auf Z; die zugehörige Klasseneinteilung sei Π_k. Nach *Bem. 1* ist Π_{k+1} eine Verfeinerung von Π_k.
Nach Lemma (3) erhält man Π_{k+1} aus Π_k, indem man gerade die Zustände einer Klasse von Π_k zusammenfasst, deren Folgezustände in der gleichen Klasse von Π_k liegen (für jedes Eingabezeichen). Π_1 ist sofort (s. *Bem. 3*) aus einer Tabelle abzulesen.
Bleibt die Frage: Terminiert diese Prozedur?
Wegen der Endlichkeit von Z ist sicher für ein k $\Pi_k = \Pi_{k+1}$.

Behauptung. Wenn $\Pi_k = \Pi_{k+1}$, so $\Pi_k = \Pi_{k+1} = \Pi_{k+2} = \ldots = \Pi$.

Beweis. Es genügt zu zeigen, dass $\Pi_k = \Pi_{k+1} \Rightarrow \Pi_{k+1} = \Pi_{k+2}$, d.h zu zeigen:
Wenn $\Pi_k = \Pi_{k+1}$ so $z_1 \overset{k+1}{\sim} z_2 \Longrightarrow z_1 \overset{k+2}{\sim} z_2$.

$$\begin{aligned} \text{Sei } z_1 \overset{k+1}{\sim} z_2 \quad &\overset{L3}{\Longrightarrow} \quad z_1 \overset{1}{\sim} z_2 \text{ und } f(z_1,x) \overset{k}{\sim} f(z_2,x)\ \forall x \in X \\ &\overset{\Pi_k = \Pi_{k+1}}{\Longrightarrow} \quad z_1 \overset{1}{\sim} z_2 \text{ und } f(z_1,x) \overset{k+1}{\sim} f(z_2,x)\ \forall x \in X \\ &\overset{L3}{\Longrightarrow} \quad z_1 \overset{k+2}{\sim} z_2 \end{aligned}$$

□

Beispiel (Prüfbitgenerator für Serienverarbeitung binär codierter Zeichen)**.** Eingabe seriell zu den Zeitpunkten t_1, t_2, t_3, t_4. Automat gibt zum Zeitpunkt t_4 Prüfbit aus (Ergänzung auf ungerade Anzahl von Einsen für Gesamtdarstellung des Zeichens).

Definition des Automaten: $A = (X, Y, Z, f, g)$

$$X = \{0,1\}, Y = \{-, 0, 1\}$$

- : kein Prüfbit bei t_1, t_2, t_3

0,1: Prüfbit bei t_4

$$Z = \{z_A, z_0, z_1, z_{00}, z_{01}, z_{10}, z_{11}, z_{000}, z_{001}, z_{010}, z_{100}, z_{101}, z_{110}, z_{011}, z_{111}\}$$

f, g	0	1
z_A	$z_0, -$	$z_1, -$
z_0	$z_{00}, -$	$z_{01}, -$
z_1	$z_{10}, -$	$z_{11}, -$
z_{00}	$z_{000}, -$	$z_{001}, -$
z_{01}	$z_{010}, -$	$z_{011}, -$
z_{10}	$z_{100}, -$	$z_{101}, -$
z_{11}	$z_{110}, -$	$z_{111}, -$
z_{000}	$z_A, 1$	$z_A, 0$
z_{001}	$z_A, 0$	$z_A, 1$
z_{010}	$z_A, 0$	$z_A, 1$
z_{100}	$z_A, 0$	$z_A, 1$
z_{011}	$z_A, 1$	$z_A, 0$
z_{101}	$z_A, 1$	$z_A, 0$
z_{110}	$z_A, 1$	$z_A, 0$
z_{111}	$z_A, 0$	$z_A, 1$

$$\Pi_1 = \left\{ \overbrace{\{z_A, z_0, z_1, z_{00}, z_{01}, z_{10}, z_{11}\}}^{p_{11}}, \overbrace{\{z_{000}, z_{011}, z_{101}, z_{110}\}}^{p_{12}}, \overbrace{\{z_{001}, z_{010}, z_{100}, z_{111}\}}^{p_{13}} \right\}$$

P_{ij} sei j-te Äquivalenzklasse von Π_i

	Π_1	0	1
	z_A	$z_0, (p_{11})$	$z_1, (p_{11})$
	z_0	$z_{00}, (p_{11})$	$z_{01}, (p_{11})$
	z_1	$z_{10}, (p_{11})$	$z_{11}, (p_{11})$
p_{11}	z_{00}	$z_{000}, (p_{12})$	$z_{001}, (p_{13})$
	z_{01}	$z_{010}, (p_{13})$	$z_{011}, (p_{12})$
	z_{10}	$z_{100}, (p_{13})$	$z_{101}, (p_{12})$
	z_{11}	$z_{110}, (p_{12})$	$z_{111}, (p_{13})$
	z_{000}	$z_A, (p_{11})$	$z_A, (p_{11})$
	z_{011}	$z_A, “$	$z_A, “$
p_{12}	z_{101}	$z_A, “$	$z_A, “$
	z_{110}	$z_A, “$	$z_A, “$
	z_{001}	$z_A, “$	$z_A, “$
p_{13}	z_{010}	$z_A, “$	$z_A, “$
	z_{100}	$z_A, “$	$z_A, “$
	z_{111}	$z_A, “$	$z_A, “$

$$\Pi_2 = \left\{ \overbrace{\{z_A, z_0, z_1\}}^{p_{21}}, \overbrace{\{z_{00}, z_{11}\}}^{p_{22}}, \overbrace{\{z_{01}, z_{10}\}}^{p_{23}}, \overbrace{\{z_{000}, z_{011}, z_{101}, z_{110}\}}^{p_{24}}, \overbrace{\{z_{001}, z_{010}, z_{100}, z_{111}\}}^{p_{25}} \right\}$$

	Π_2	0	1
	z_A	$z_0, (p_{21})$	$z_1, (p_{21})$
p_{21}	z_0	$z_{00}, (p_{22})$	$z_{01}, (p_{23})$
	z_1	$z_{10}, (p_{23})$	$z_{11}, (p_{22})$
p_{22}	z_{00}	$z_{000}, (p_{24})$	$z_{001}, (p_{25})$
	z_{11}	$z_{110}, (p_{24})$	$z_{111}, (p_{25})$
p_{23}	z_{10}	$z_{100}, (p_{25})$	$z_{101}, (p_{24})$
	z_{01}	$z_{010}, (p_{25})$	$z_{011}, (p_{24})$
	z_{000}	$z_A, (p_{11})$	$z_A, (p_{11})$
	z_{011}	"	"
	z_{101}	"	"
	z_{110}	"	"
	z_{001}	"	"
	z_{010}	"	"
	z_{100}	"	"
	z_{111}	"	"

$$\Pi_3 = \left\{ \overbrace{\{z_A\}}^{p_{31}}, \overbrace{\{z_0\}}^{p_{32}}, \overbrace{\{z_1\}}^{p_{33}}, \overbrace{\{z_{00}, z_{11}\}}^{p_{34}} . \overbrace{\{z_{01}, z_{10}\}}^{p_{35}}, \overbrace{\{z_{000}, z_{011}, z_{101}, z_{110}\}}^{p_{36}}, \overbrace{\{z_{001}, z_{010}, z_{100}, z_{111}\}}^{p_{37}} \right\}$$

	Π_3	0	1
p_{31}	z_A	z_0	z_1
p_{32}	z_0	z_{00}	z_{01}
p_{33}	z_1	z_{10}	z_{11}
p_{34}	z_{00}	$z_{000}, (p_{36})$	$z_{001}, (p_{37})$
	z_{11}	$z_{110}, (p_{36})$	$z_{111}, (p_{37})$
p_{35}	z_{01}	$z_{100}, (p_{37})$	$z_{101}, (p_{36})$
	z_{10}	$z_{010}, (p_{37})$	$z_{011}, (p_{35})$
p_{36}	z_{000}	"	"
	z_{011}	"	"
	z_{101}	"	"
	z_{110}	"	"
p_{37}	z_{001}	"	"
	z_{010}	"	"
	z_{100}	"	"
	z_{111}	"	"

$$\Pi_4 = \Pi_3 = \Pi$$

A_{R}:

$f_{\mathrm{R}}, g_{\mathrm{R}}$	0	1
$\overline{z_A}$	$\overline{z_0}, -$	$\overline{z_1}, -$
$\overline{z_0}$	$\overline{z_{00}}, -$	$\overline{z_{01}}, -$
$\overline{z_1}$	$\overline{z_{01}}, -$	$\overline{z_{00}}, -$
$\overline{z_{00}}$	$\overline{z_{000}}, -$	$\overline{z_{001}}, -$
$\overline{z_{01}}$	$\overline{z_{001}}, -$	$\overline{z_{000}}, -$
$\overline{z_{000}}$	$\overline{z_A}, 1$	$\overline{z_A}, 0$
$\overline{z_{001}}$	$\overline{z_A}, 0$	$\overline{z_A}, 1$

Reduktion von A mit 15 Zuständen auf A_{R} mit 7 Zustände.

1.1.3 MOORE-Automaten

Definition. Ein MOORE-Automat ist ein 5-Tupel $A = (X, Y, Z, f, h)$ mit nichtleeren endlichen Mengen X, Y, Z und Abbildungen (X, Y, Z wie bei MEALY!):

$$\begin{aligned} f &: Z \times X \to Z \\ h &: Z \to Y \end{aligned}$$

Beim MOORE-Automat ist die Ausgabefunktion nicht **direkt** von der Eingabe abhängig. Bei der Darstellung in Tabellenform benötigt man für die Darstellung von h nur eine Spalte:

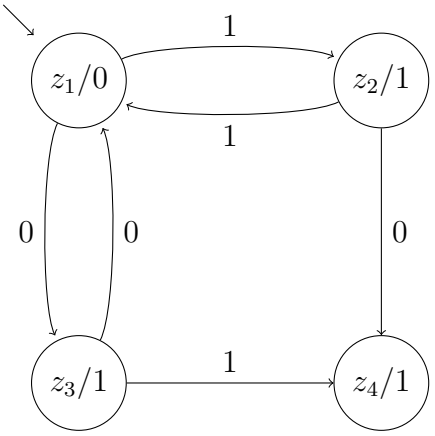
	x_1	$\dots$	x_n	h
z_1	$f(z_1, x_1)$	$\dots$	$f(z_1, x_n)$	$h(z_1)$
$\vdots$	$\vdots$	$\ddots$		$\vdots$
z_k	$f(z_k, x_1)$	$\dots$	$f(z_k, x_n)$	$h(z_k)$

Bei der Darstelluung mittels gerichteter Graphen wird das Ausgabezeichen in die Knotenbeschriftung einbezogen und die Kanten werden nur mit den Eingabezeichen beschriftet.

Anfangszustand z setzen
Auf Ausgabeband $y = h(z)$ schreiben
Leseband auf 1. Zeichen der Eingabe setzen
`while`: Feld unter Lesevorrichtung nicht leer `do` - Zeichen X unter Lesekopf lesen - Zustand $z := f(z, x)$ - Ausgabeband um ein Feld weiterrücken - Auf Ausgabeband $y = h(z)$ schreiben - Eingabeband um ein Feld weiterrücken

Insbesondere wird auch bei eingegebenem leeren Wort ε ein Zeichen ausgegeben; bei Eingabe eines Strings der Länge n hat der Outputstring die Länge $n + 1$, wobei das erste Zeichen (des Outputstrings) i.A. nicht relevant ist.

Beispiel.



	0	1	h
z_1	z_3	z_2	0
z_2	z_4	z_1	1
z_3	z_1	z_4	1
z_4	z_4	z_4	1

mit $X = Y = \{0, 1\}$

Eingabe:	ε	Ausgabe:	0
Eingabe:	0	Ausgabe:	01
Eingabe:	011	Ausgabe:	0111
Eingabe:	110	Ausgabe:	0101

Dementsprechend ist auch die erweiterte Ergebnisfunktion wie folgt zu definieren:

$$h^* : Z \times X^* \longrightarrow Y^*$$

$$h^*(z, \varepsilon) = h(z),\ h^*(z, px) = h^*(z, p)h(f^*(z, px))$$

und

$$h_z^* : X^* \longrightarrow Y^* \text{ mit } h_z^*(p) = h^*(z, p)$$

Alle Ergebnisse und Definitionen für MEALY-Automaten finden dann ihre Entsprechung bei MOORE-Automaten. Bei der Reduktion hat allerdings bei MOORE-Automaten der Algorithmus mit 0-äquivalenten Zuständen zu beginnen, d.h. mit Zuständen, die bei Eingabe von ε die gleiche Ausgabe erzeugen:
Für das Beispiel wäre also $\Pi_0 = \{\{z_1\}, \{z_2, z_3, z_4\}\}$.

Zusammenhang zwischen MEALY- und MOORE-Automaten
Wegen der verschiedenen Länge der Ausgabestrings bei gleichem Eingabestring bei MEALY- und MOORE-Automaten kann ein MOORE-Automat nicht zu einem MEALY-Automaten äquivalent sein.
Es stört dabei das (redundante) 1. ausgegebene Zeichen.

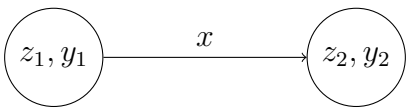
Definition. Die *beschränkte Leistung* $\overline{h_z^*}$ eines Zustandes z eines MOORE-Automaten (wie oben) ist definiert durch

$$h^*(z, \varepsilon)\ \overline{h_z^*}(p) = h_z^*(p) \quad \forall p \in X^*$$

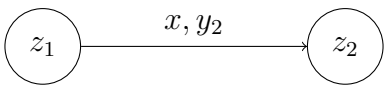
(d.h. $\overline{h_z^*}(p)$ ist $h_z^*(p)$, wobei 1. Zeichen gestrichen wird)

Definition. Der MEALY-Automat $A_1 = (X, Y, Z_1, f_1, g_1)$ und der MOORE-Automat $A_2 = (X, Y, Z_2, f_2, h_2)$ heißen *gleichwertig*, wenn die Menge der Leistungen von A_1 mit der Menge der beschränkten Leistungen von A_2 übereinstimmt.
Offenbar gibt es zu jedem MOORE-Automaten $A = (X, Y, Z, f, h)$ einen gleichwertigen MEALY-Automaten $A' = (X, Y, Z', f', g')$.

Anschaulich: Der Graph von A geht in den von A' über, indem jeder Teil



ersetzt wird durch



Formal:

$$\begin{aligned} Z' &:= Z \\ f' &:= f \\ g'(z,x) &:= h(f(z,x)) \end{aligned}$$

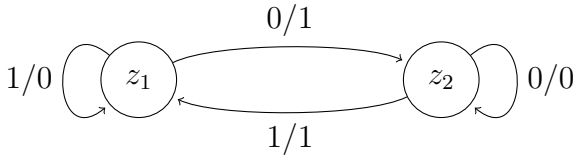
siehe Bsp.

Frage. Können MEALY-Automaten echt mehr leisten?

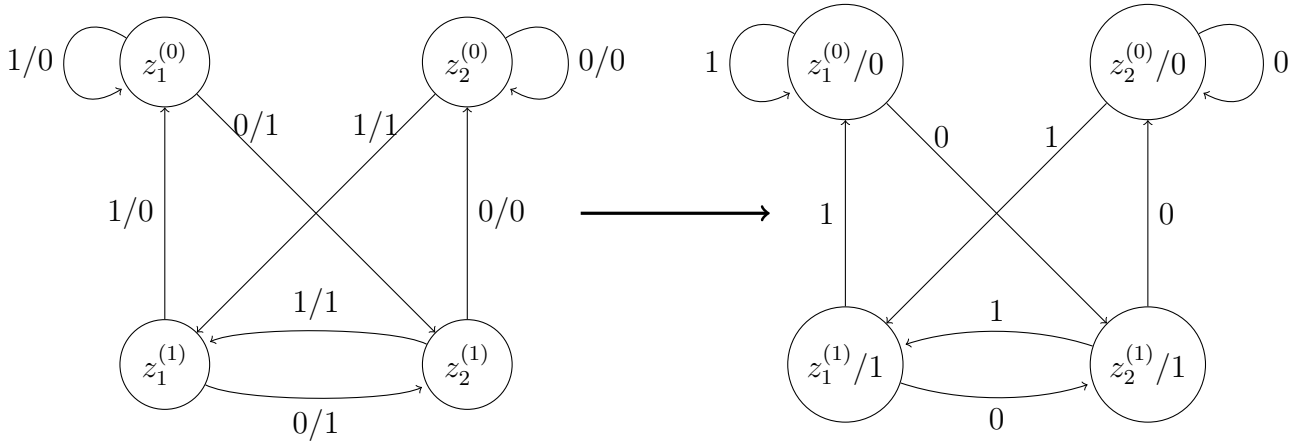
Antwort. Nein!

Idee. Konstruktion umkehren?

Geht im „*Beispiel*“, aber:



Trick: Wir „*verdoppeln*“ hier die Zustände und die Kanten, wobei alle Kanten mit der Ausgabe 0 in die 0-Komponente führen und analog alle Kanten mit Ausgabe 1 in 1-Komonenten.



Satz 1.3. Zu jedem MEALY-Automaten $A = (X, Y, Z, f, g)$ gibt es einen gleichwertigen MOORE-Automaten $A' = (X, Y, Z', f', h')$.

Beweis.

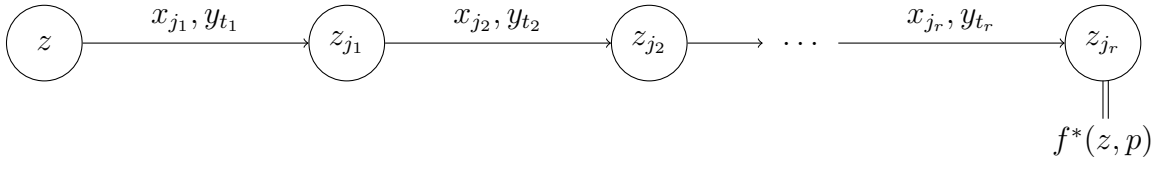
$$Z' = Z \times Y,\ f'((z,y),x) := (f(z,x),\ g(z,x)) \quad \forall x \in X, y \in Y, z \in Z$$

$$h'(z,y) := y \quad \forall x \in X, y \in Y, z \in Z$$

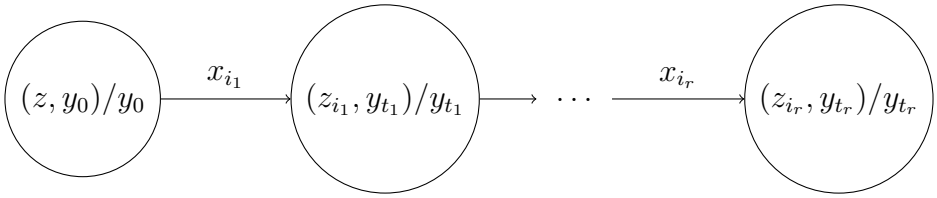
Sei $y_0 \in Y$ beliebig. Es genügt zu zeigen, dass

$$(\#) \qquad g^*(z,p) = \overline{h'^*}\,((z,y_0),p) \quad \forall p \in X^*, z \in Z$$

Sei $p = x_{i_1} \dots x_{i_r} \in X^*$
Man hat

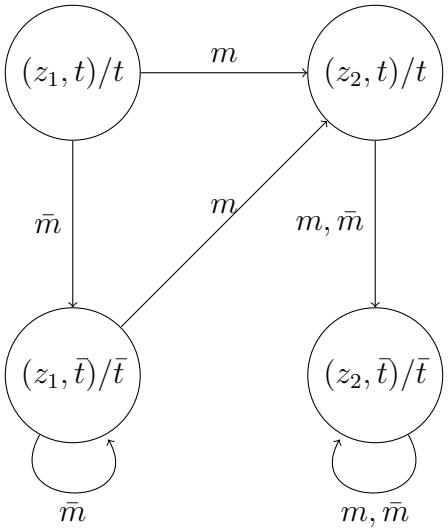


mit $g^*(z,p) = y_{t_1} y_{t_2} \dots y_{t_r}$
Nach Definition von f' und h' wird



in A', d.h. $h'^*((z,y_0),p) = y_0 y_{t_1} y_{t_2} \dots y_{t_r}$
also richtig. □

Beispiel (Mausefalle ohne Speck)**.**



1.2 Endliche Automaten (ohne Ausgabe)

1.2.1 Deterministische endliche Automaten

Beispiel. PASCAL-Syntax für Zahlen hat folgende Form:

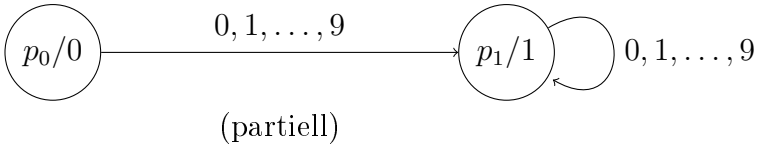
$$
\begin{aligned}
\langle \text{number} \rangle &::= \langle \text{unsigned number} \rangle | \langle \text{sign} \rangle \langle \text{unsigned number} \rangle \\
\langle \text{unsigned number} \rangle &::= \langle \text{unsigned integer} \rangle | \langle \text{unsigned real} \rangle \\
\langle \text{unsigned real} \rangle &::= \langle \text{unsigned integer} \rangle . \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \} \\
&\quad | \langle \text{unsigned integer} \rangle . \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \} \, \text{E} \langle \text{scale factor} \rangle \\
&\quad | \langle \text{unsigned integer} \rangle \text{E} \langle \text{scale factor} \rangle \\
\langle \text{scale factor} \rangle &::= 0|1|2|3|4|5|6|7|8|9 \\
\langle \text{sign} \rangle &::= +|-
\end{aligned}
$$

Aufgabe. Man konstruiere einen MOORE-Automaten A mit $Y = \{0,1\}$, der erkennt, ob eine eingegebene Zeichenfolge eine korrekte PASCAL-Zahl ist, d.h. er soll 1 ausgeben, wenn das der Fall ist und sonst 0.

Zunächst betrachten wir

$$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \}$$

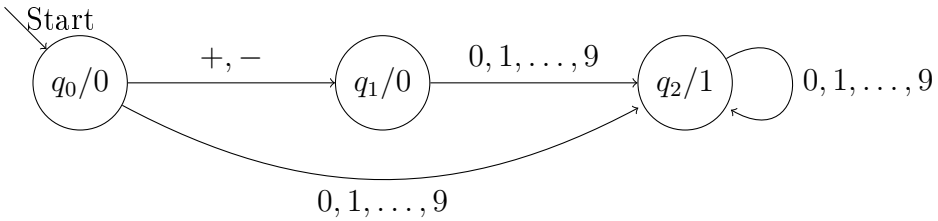
$$\langle \text{digit} \rangle ::= 0|1|2|3|4|5|6|7|8|9$$



(partiell)

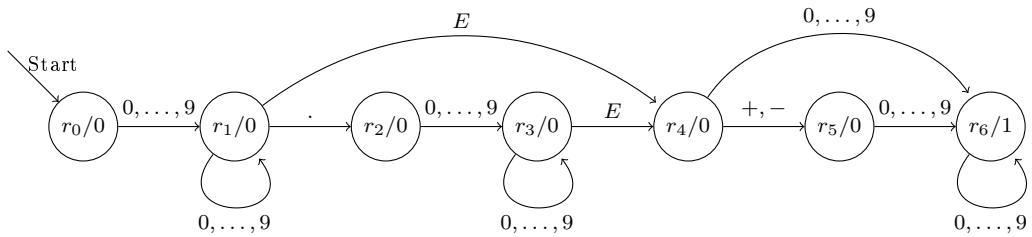
leistet das Gewünschte (wenn in p_0 gestartet).
Definition des Exponenten:

$$\begin{aligned} \langle \text{scale factor} \rangle &::= \langle \text{unsigend integer} \rangle | \langle \text{sign} \rangle \langle \text{unsigned integer} \rangle \\ \langle \text{sign} \rangle &::= +|- \end{aligned}$$

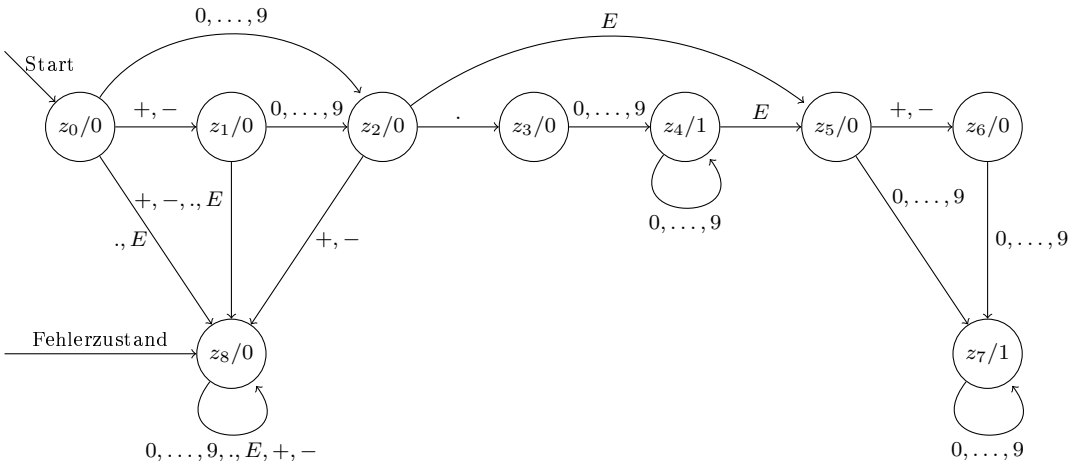


Startzustand q_0:

$$\begin{aligned} \langle \text{unsigned real} \rangle ::= \; & \langle \text{unsigend integer} \rangle . \langle \text{digit} \rangle \, \{ \langle \text{digit} \rangle \} \, | \\ & \langle \text{unsigned integer} \rangle . \langle \text{digit} \rangle \, \{ \langle \text{digit} \rangle \} \, \text{E} \langle scalefactor \rangle | \\ & \langle \text{unsigend integer} \rangle E \langle \text{scale factor} \rangle \end{aligned}$$



$$\begin{aligned} \langle \text{number} \rangle &::= \langle \text{unsigned number} \rangle | \langle \text{sign} \rangle \langle \text{unsigned number} \rangle \\ \langle \text{unsigned number} \rangle &::= \langle \text{unsigned integer} \rangle | \langle \text{unsigned real} \rangle \end{aligned}$$



$X = \{0, \ldots, 9, +, -, ., E\}$
Dabei in z_0 gestartet. z_8 ist Fehlerzustand.

Aufgabe damit gelöst.

Man verzichtet nun gewöhnlich auf die Ausgabe der Zeichen 0 bzw. 1 und charakterisiert stattdessen die Zustände, bei denen 1 ausgegeben wird, als *Endzustände*;
auch wird ein Endzustand ausgezeichnet.

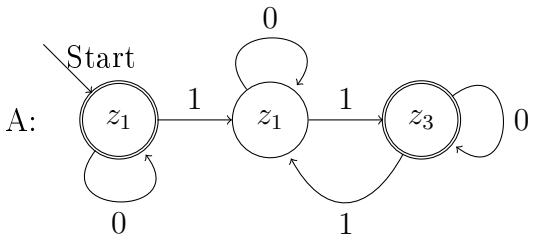
Definition. Ein endlicher deterministischer Automat (DA) ist ein 5-Tupel $A = (X, Z, f, z_A, Z_E)$, wobei X eine endliche Menge $\neq \varnothing$ - das Eingabealphabet, Z eine endliche Menge $\neq \varnothing$ - die Menge der Zustände $f : Z \times X \longrightarrow Z$ die Zustandsüberführungsfunktion, $z_A \in Z$ der Anfangszustand von A und $Z_E \subseteq Z$, die Menge der Endzustände ist.

Definition. Die von einem DA $(A = X, Z, f, z_A, Z_E)$ *akzeptierte Sprache* $L(A)$ ist die Menge aller $p \in X^*$, für die $f^*(z_A, p) \in Z_E$.

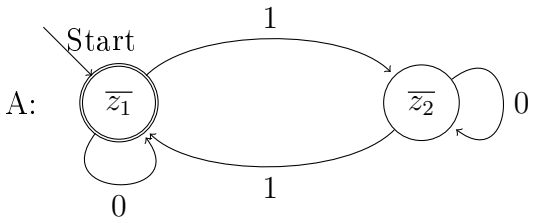
Beispiel. Das obige Beispiel zeigt einen DA A mit $X = \{0, \ldots, 9, +, -, ., E\}$,
$Z = \{z_0, z_1, \ldots, z_8\}, z_A = z_0$ und $Z_E = \{z_2, z_4, z_7\}$, f ist durch den Graphen gegeben. $L(A)$ ist Menge aller „PASCAL-*Zahlen*“.

Bemerkung. Ein DA kann wie im Beispiel als MOORE-Automat mit $Y = \{0, 1\}$ und $h(z) = 1$ g.d.w. $z \in Z_E$ aufgefasst werden.
In diesem Sinne sind Äquivalenz etc. für DA's erklärt.
Insbesondere kann man endliche Automaten reduzieren (und zusätzlich vom Anfangszustand nicht erreichbare Zustände entfernen).

Beispiel. Es soll ein Akzeptor konstruiert werden für die Strings aus $\{0, 1\}^*$, die eine gerade Anzahl von Einsen enthalten.
Mögliche Lösung:



A ist reduzierbar: Die Zustände z_1 und z_3 sind äquivalent, wie man sofort sieht.
Reduzierter Automat:



Beispiel. Sei $X = \{0, 1\}, p \in X^*$ heißt Palindrom, wenn die Umkehrung von p gleich p ist.

Frage. Gibt es einen Akzeptor für alle Palindrome aus X^*?

Antwort. Nein! (wird später gezeigt)

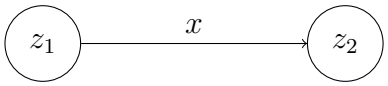
Allgemeine Frage. Welche Mengen aus X^* sind so beschaffen, dass sie von einem DA akzeptiert werden?

1.2.2 Nichtdeterministische endliche Automaten

Definition. Ein nichtdeterministischer Automat (NDA) ist ein 5-Tupel $A = (X, Z, R_f, Z_A, Z_E)$, wobei X, Z, Z_E wie bei DA gegeben sind, $\varnothing \neq Z_A \subseteq Z$ ist Menge der Startzustände und $R_f \subseteq Z \times X \times Z$ eine Übergangsrelation ist. (oder $R_f : Z \times X \longrightarrow \wp(Z)$)

Bemerkung.

1. Ist $(z_1, x, z_2) \in R_f$, so wird dies im Graphen von A durch



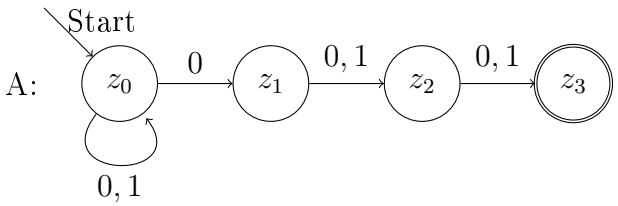
   dargestellt.

2. DA's sind Spezialfälle von NDA's, gerade dann nämlich, wenn R_f eine Abbildung von $Z \times X$ in Z ist und $|Z_A| = 1$.

Definition. Die vom NDA $A = (X, Z, R_f, Z_A, Z_E)$ akzeptierte Sprache $L(A)$ ist $\{p \in X^* : \exists$ Folge $(z_{j_0}, x_{j_1}, z_{j_1}), \ldots, (z_{j_{r-1}}, x_{j_r}, z_{j_r})\}$ von Elementen aus R_f mit $z_{j_0} \in Z_A, z_{j_r} \in Z_E$ und $p = x_{j_1} x_{j_2} \ldots x_{j_r}\}$.

Also: $p \in X^*$ wird von A akzeptiert g.d.w. es eine mögliche Folge von Übergängen von einem Anfangszustand in einen Endzustand ermöglicht.

Beispiel. $X = \{0, 1\}$. Wir konstruieren einen Akzeptor für alle $p \in X^*$, deren drittletztes Zeichen eine 0 ist.
Die Lösung für NDA ist einfach:



Hier:

$$\begin{aligned} X &= \{0,1\}, Z = \{z_0, z_1, z_2, z_3\} \\ R_f &= \{\{z_0, 0, z_0\}, \{z_0, 0, z_1\}, \{z_0, 1, z_0\}, \{z_1, 0, z_2\}, \{z_1, 1, z_2\}, \{z_2, 0, z_3\}, \{z_2, 1, z_3\}, \} \\ Z_A &= \{z_0\}, Z_E = \{z_3\} \\ L &= \{p \in \{0,1\}^*, p = p_1 0xy, p_1 \in \{0,1\}^*\} \end{aligned}$$

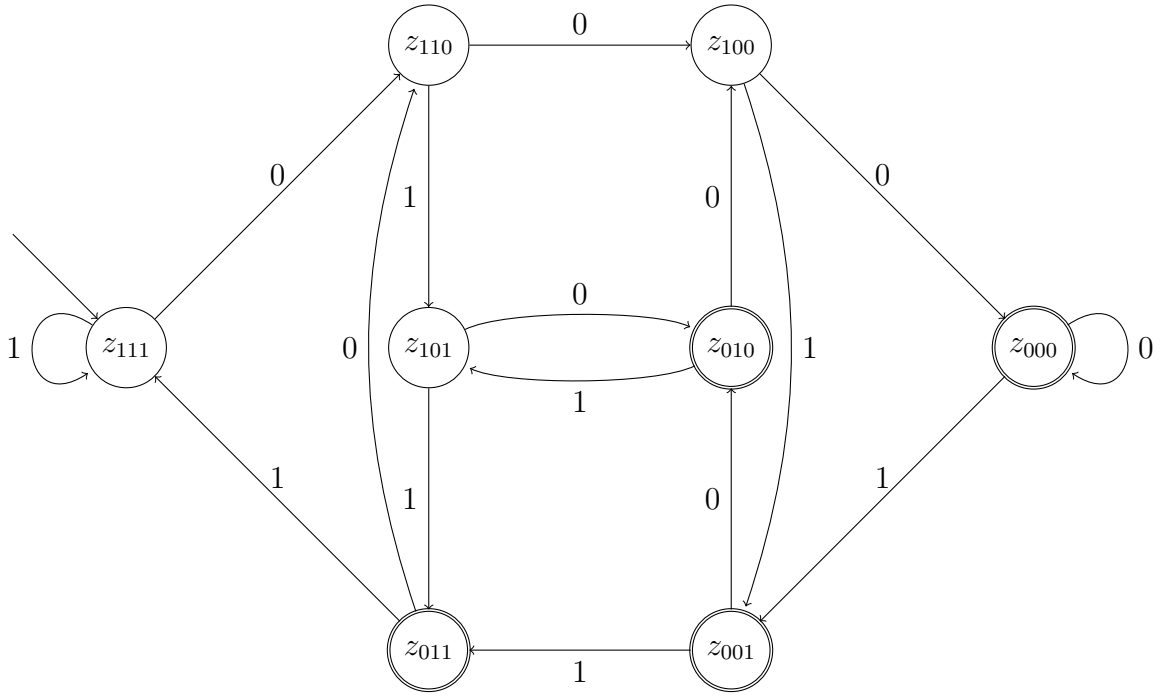
Behauptung. $L = L(A)$

1. $L \subseteq L(A)$ ✓

2. $L(A) \subseteq L$
   Sei $p \in L(A) \Longrightarrow |p| \geq 3$
   Ang.: $p = p_1 1xy$. Aber dann ist kein Übergang von z_0 nach z_2 möglich $\longrightarrow$ ↯

Frage. Gibt es ein DA „*für diese Sprache*"?

Antwort. Ja! Da es nur auf die letzten 3 Zeichen ankommt, wählen wir Zustände $z_{000}, \ldots, z_{111}$. Wir können mit diesen 8 Zuständen auskommen, wenn wir z_{111} als Startsequenz wählen, da dann 3 Eingabezeichen nötig sind, um einen der Endzustände $z_{000}, z_{001}, z_{011}$ zu bekommen.



Wenn $\mathcal{L}_{Det}$ die Klasse der von DAs akzeptierten Sprachen ist und $\mathcal{L}_{NDet}$ die Klasse der von NDAs akzeptierten Sprachen ist, so gilt trivialerweise

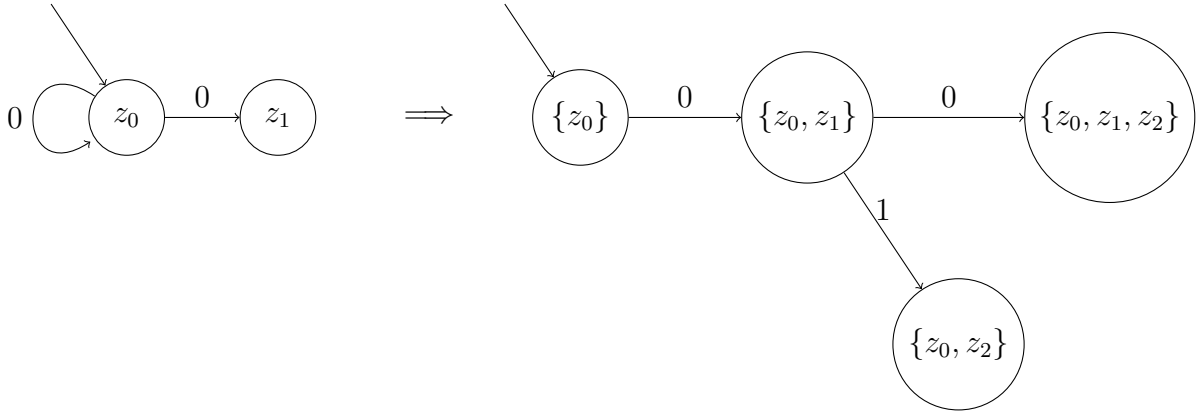
$$\mathcal{L}_{Det} \subseteq \mathcal{L}_{NDet}$$

Gilt $\mathcal{L}_{Det} \subsetneqq \mathcal{L}_{NDet}$? Nein!

Satz 1.4. Es sei A ein NDA. Dann gibt es einen DA A' mit $L(A) = L(A')$.

Beweis. Sei $A = (X, Z, R_f, Z_A, Z_E)$. Wir definieren $A' = (X, Z', f', z'_a, Z'_E)$ wie folgt:

$$\begin{array}{rcl} Z' & := & \mathcal{R}(Z)(= \{Q : Q \subseteq Z\}) \text{ (Potenzmenge von } Z) \\ f' & : & Z' \times X \longrightarrow Z \text{ wird def. durch } f'(Q, x) := Q' \\ Q' & = & \{z' \in Z : \exists z \in Q : (z, x, z') \in R_f\} \end{array}$$



$$z'_A := Z_A,\ Z'_E := \{Q : Q \cap Z_E \neq \varnothing\}$$

Sei $p = x_{j_1} \dots x_{j_r} \in X^*$.
a) $p \in L(A')$, d.h. $\exists$ Folge von Teilmengen $Q_{j_0}, Q_{j_1}, \dots, Q_{j_r}$;

$Q_{j_0} = Z_A$, $Q_{j_r} \in Z'_E$ mit $f'(Q_{j_i}, x_{j_{(i+1)}}) = Q_{j_{(i+1)}} \quad (i = 0, \ldots, r-1)$

$$\begin{aligned} \Longrightarrow \exists z_{j_r} \in Q_{j_r} \cap Z_E \quad , \quad & z_{j_{(r-1)}} \in Q_{j_{(r-1)}} : (z_{j_{(r-1)}}, x_{j_r}, z_{j_r}) \in R_f \\ \Longrightarrow \quad & \exists z_{j_{(r-2)}} \in Q_{j_{(r-2)}} : (z_{j_{(r-2)}}, x_{j_{(r-1)}}, z_{j_{(r-1)}}) \in R_f \\ & \ldots \\ & \exists z_{j_0} \in Q_{j_0} = Z_A : (z_{j_0}, x_{j_1}, z_{j_1}) \in R_f \\ \Longrightarrow \quad & p \in L(A) \end{aligned}$$

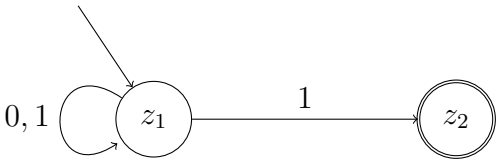
b) $p \in L(A) \Longrightarrow \exists z_{j_0}, z_{j_1}, \ldots, z_{j_r}$ mit $(z_{j_0}, x_{j_1}, z_{j_1}) \in R_f, \ldots, (z_{j_{(r-1)}}, x_{j_r}, z_{j_r}) \in R_f$
$z_{j_0} \in Z_A, z_{j_r} \in Z_E$

$$f'(Z_A, x_{j_1}) =: Q_{j_1} \ni z_{j_1}, f'(Q_{j_1}, x_{j_1}) =: Q_{j_2} \ni z_{j_2}, \ldots, Q_{j_r} \ni z_{j_r} \Longrightarrow Q_{j_r} \in Z'_E$$
$$\Longrightarrow p \in L(A')$$

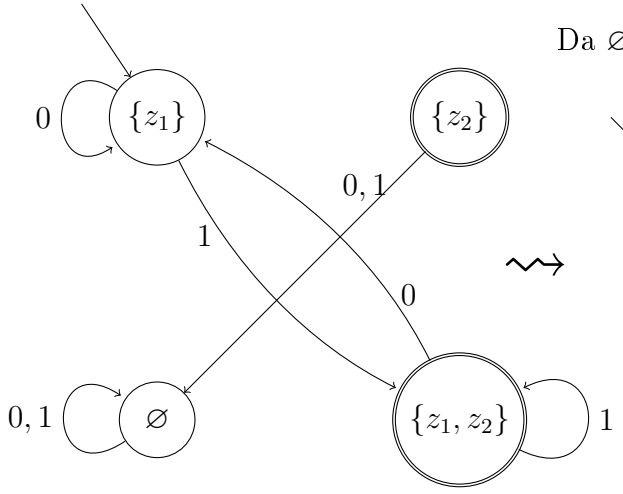
□

Bemerkung. Ist $|Z| = k$, so besitzt der konstruierte DA 2^k Zustände. Häufig kann man welche weglassen bzw. reduzieren.

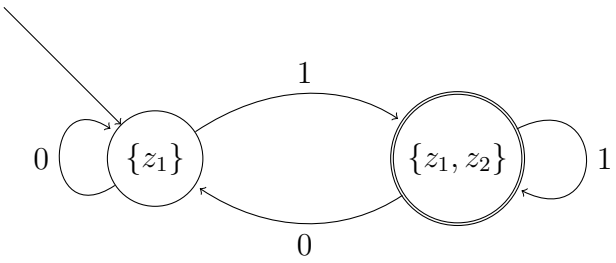
Beispiel. Akzeptor für alle $p \in \{0,1\}^*$, die mit 1 enden.



DA nach Konstruktion:



Da $\varnothing$ und $\{z_2\}$ nicht erreichbar sind (von $\{z_1\}$ aus):



1.3 Formale Sprachen

Geläufig ist die Backus-Naur-Form der Beschreibung einer Programmiersprache (BNF).
Als vereinfachtes Beispiel betrachten wir die Menge der Ausdrücke einer imaginären Programmiersprache mit nur 2 Variablen a und b, Zahlkonstanten $0, 1, \ldots, 9$ und $+, -, *, /$ als Operatoren:

1. $\langle \text{Ausdruck} \rangle ::= \langle \text{Variable} \rangle \mid \langle \text{Konstante} \rangle \mid (\langle \text{Ausdruck} \rangle \langle \text{Operator} \rangle \langle \text{Ausdruck} \rangle)$
2. $\langle \text{Konstante} \rangle ::= 0|1|2|3|4|5|6|7|8|9$
3. $\langle \text{Variale} \rangle ::= a|b$
4. $\langle \text{Operator} \rangle ::= +| - | * |/$

Jede dieser Regeln ist als eine *Ersetzungsregel* (Produktion) aufzufassen.
Die 1. Regel besagt z.B., dass $\langle$Ausdruck$\rangle$ ersetzt werden kann durch $\langle$Variable$\rangle$ oder $\langle$Konstante$\rangle$ oder $(\langle$Ausdruck$\rangle \langle$Operator$\rangle \langle$Ausdruck$\rangle)$.
Die Zeichen $::=$ und $|$ sind sogenannte metasprachliche Zeichen.
Ferner existieren *Nichtterminale* (auch syntaktionale Variablen genannt), diese sind in spitze Klammern eingeschlossen.
Die übrigen Zeichen $0, \ldots, 9, +, -, *, /, (,), a, b$ heißen *Terminale*.
Ferner wird ein Nichtterminal als Startsymbol ausgezeichnet - hier $\langle$Ausdruck$\rangle$.
Dann sind erlaubte Ausdrücke (in dieser „*Sprache*“) alle Zeichenfolgen aus Terminalen, die mittels der Regeln 1. bis 4. aus $\langle$Ausdruck$\rangle$ mittels Ersetzung hergeleitet werden können, z.B.

$$\begin{aligned}
\langle \text{Ausdruck} \rangle &\overset{1.}{\Longrightarrow} (\langle \text{Ausdruck} \rangle \langle \text{Operator} \rangle \langle \text{Ausdruck} \rangle) \overset{1.}{\Longrightarrow} (\langle \text{Variable} \rangle \langle \text{Operator} \rangle \langle \text{Ausdruck} \rangle) \\
&\overset{3.}{\Longrightarrow} (a \langle \text{Operator} \rangle \langle \text{Ausdruck} \rangle) \overset{4.}{\Longrightarrow} (a + \langle \text{Ausdruck} \rangle) \\
&\overset{1.}{\Longrightarrow} (a + (\langle \text{Ausdruck} \rangle \langle \text{Operator} \rangle \langle \text{Ausdruck} \rangle)) \\
&\overset{1.}{\Longrightarrow} (a + (\langle \text{Variable} \rangle \langle \text{Operator} \rangle \langle \text{Ausdruck} \rangle)) \overset{3.}{\Longrightarrow} (a + (a \langle \text{Operator} \rangle \langle \text{Ausdruck} \rangle)) \\
&\overset{4.}{\Longrightarrow} (a + (a * \langle \text{Ausdruck} \rangle)) \overset{1.}{\Longrightarrow} (a + (a * \langle \text{Variable} \rangle)) \overset{3.}{\Longrightarrow} (a + (a * b))
\end{aligned}$$

Definition. Ein Ersetzungssystem (oder *Semi-Thue-System*) ist ein Paar (Σ, P) mit

- Σ ist ein Alphabet
- $P \subset \Sigma^* \times \Sigma^*$ ist eine endliche Menge;
  Elemente von P heißen *Produktionen* oder *Regeln*; für $(\alpha, \beta) \in P$ schreibt man $\alpha \longrightarrow \beta$ („*oben*“ $\alpha ::= \beta$)

Definition (*Arbeitsweise* des Ersetzungsystems $(\Sigma, P) =: E$)**.** Die Anwendung einer Regel $\alpha \longrightarrow \beta$ auf $p \in \Sigma^*$ besteht darin, dass in p das Teilwort α (an einer beliebigen Stelle) durch β ersetzt wird, man schreibt
$p \Longrightarrow p'$, d.h.
$p \Longrightarrow p'$ g.d.w. $\exists x, y \in \Sigma^*, \alpha \longrightarrow \beta \in P$, so dass $p = x\alpha y, p' = x\beta y$.
Man sagt auch p' sei *direkt ableitbar* (vermöge P) aus p.
p' heißt *ableitbar* aus $p, p \overset{*}{\Longrightarrow} p'$, g.d.w. $p = p'$
oder es gibt eine Folge $p_1, \ldots, p_n \in \Sigma^*$ mit $p = p_1, p_1 \Longrightarrow p_2, p_2 \Longrightarrow p_3, \ldots, p_{n-1} \Longrightarrow p_n = p'$.
($\overset{*}{\Longrightarrow}$ ist die reflexive transitive Hülle der Relation $\Longrightarrow$)

Definition. Ein 4-Tupel $G = (N, T, S, P)$ heißt Grammatik (oder CHOMSKY-Grammatik), wenn

- N ist ein Alphabet (Menge von Nichtterminalen)
- T ist ein Alphabet (Menge von Terminalen) mit $N \cap T = \varnothing$
- $(N \cup T, P)$ ist ein Ersetzungssystem
- $S \in N$ (Startzeichen)

- Es gibt in P keine Regel der Form $\varepsilon \longrightarrow \alpha$

Da ein Ersetzungssystem vorliegt, sind $\Longrightarrow$ und $\overset{*}{\Longrightarrow}$ definiert.

Definition. Ist $G = (N, T, S, P)$ eine Grammatik, so heißt $L(G) = \left\{ v \in T^* : S \overset{*}{\Longrightarrow} v \right\}$

Beispiel.

1. $N = \{S\}, T = \{a\}, P = \{S \to \varepsilon, S \to aS\}$

$$\begin{array}{ccccccc} S & \Longrightarrow & aS & \Longrightarrow & aaS & \Longrightarrow & \dots \\ \Downarrow & & \Downarrow & & \Downarrow & & \\ \varepsilon & & a & & aa = a^2 & & \dots \end{array}$$

$\Longrightarrow L(G) = \{a^i : i \geq 0\} = T^*$

2. $N = \{S\}, T = \{a, b\}, P = \{S \to \varepsilon, S \to aSb\}$

$$\begin{array}{ccccccc} S & \Longrightarrow & aSb & \Longrightarrow & aaSbb & \Longrightarrow & \dots \\ \Downarrow & & \Downarrow & & \Downarrow & & \\ \varepsilon & & ab & & a^2b^2 & & \dots \end{array}$$

$\Longrightarrow L(G) = \{a^i b^i : i \geq 0\}$

Für $L(G) = \{a^i b^i : i \geq 1\}$
$S \to aSb, S \to ab$

3. $G = (\{S\}, \{a, b\}, \{S \to \varepsilon, S \to aSb, S \to bSa, S \to SS\}, S)$

Behauptung. $L(G) = \{w \in \{a, b\}^* : w \text{ enthält gleichviele } a \text{ und } b\} =: L$

Beweis.

(a) $L(G) \subseteq L$, weil a und b in den Produktionen nur paarweise auftreten.

(b) $L \subseteq L(G)$. Sei $w \in L$. Wir beweisen $w \in L(G)$ mit Induktion nach der halben Länge i von w.
$i = 0, w = \varepsilon$; richtig, weil $S \to \varepsilon \in P$.
Angenommen, die Behauptung gelte für Wörter mit halber Länge $\leq i$.
Sei $|w| = 2(i + 1)$.

i. w beginnt mit a und endet mit b, d.h.

$$w = aw'b \Longrightarrow w' \in L$$

Da $|w'| = 2i$, ist nach Induktionsvoraussetzung $w' \in L(G)$, d.h. $S \overset{*}{\Longrightarrow} w'$. Dann $S \Longrightarrow L(G)$.

ii. $w = bw'a$ analog

iii. $w = aw'a$ (bzw. $w = bw'b$)

Dann gilt $(*)$ $\begin{cases} w = w_1 w_2; w_1, w_2 \in L \\ |w_1|, |w_2| \leq 2i \end{cases}$

Nach Induktionsvoraussetzung ist $w_1, w_2 \in L(G)$, also $S \overset{*}{\Longrightarrow} w_1, S \overset{*}{\Longrightarrow} w_2$.
Folglich $S \Longrightarrow SS \overset{*}{\Longrightarrow} w_1 w_2 = w$, d.h. $w \in L(G)$.
(Bew. von $(*)$: Sei $f(j)$ die Anzahl der a's.
Anzahl der f's in j-ten Präfix von w: $f(1) = 1, f(2i + 1) = -1$
Also $\exists$ Präfix w_1 mit $f = 0$.)

□

4. $N = \{S, X, Y\}, T = \{a, b, c\}$,
   $P = \{S \to abc, S \to aXbc, Xb \to bX, Xc \to Ybcc, bY \to Yb, aY \to aaX, aY \to aa\}$

$$\begin{array}{ccccccccccc} S & \Longrightarrow & aXbc & \Longrightarrow & abXc & \Longrightarrow & abYbcc & \Longrightarrow & aYb^2c^2 & \Longrightarrow & a^2Xb^2c^2 \\ \Downarrow & & & & & & & & \Downarrow & & \\ abc & & & & & & & & a^2b^2c^2 & & \end{array}$$

   $L(G) = \{a^i, b^i, c^i : i \geq 1\}$

Definition. Zwei Grammatiken heißen äquivalent, wenn sie die gleiche Sprache erzeugen. (G_1 und G_2 äquivalent g.d.w. $L(G_1) = L(G_2)$)

Klassifikation der Grammatik nach Chomski
Eine Grammatik $G = (N, T, S, P)$ mit $V := N \cup T$ heißt

- *vom Typ 0*, wenn keine Einschränkung an P existiert
- *vom Typ 1* oder *kontextsensitiv*, wenn jede Produktion von P von der Form ist $q_1Xq_2 \longrightarrow q_1wq_2$ mit $q_1, q_2, w \in V^*, X \in N, w \neq \varepsilon$ oder $S \to \varepsilon$ ist. Kommt $S \to \varepsilon$ in P vor, so darf S auf keiner rechten Seite einer Produktion von P vorkommen (im „*Kontext*“ q_1, q_2 kann X durch w ersetzt werden)
- *vom Typ 2* oder *kontextfrei*, wenn jede Produktion von P von der Form $X \to w$ mit $X \in N, w \in V^*$ ist
- *vom Typ 3* oder (rechts)*regulär*, wenn jede Produktion von P von der Form $X \to wY$ oder $X \to w$ mit $X, Y \in N, w \in T^*$ ist.

Definition. Eine Sprache heißt vom Typ i ($i = 0, 1, 2, 3$), wenn sie von einer Grammatik vom Typ i erzeugt wird. (d.h. erzeugt werden kann!)

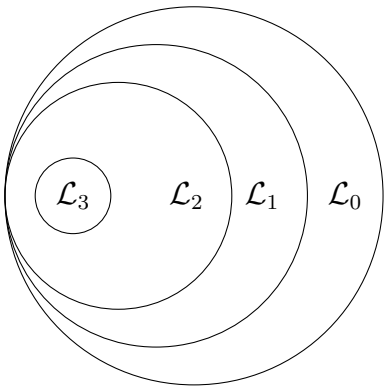
Beispiel. Grammatik $G : S \to cXd, X \to cX|\varepsilon, X \to Xd; L(G) = \{c^id^j : i, j \geq 1\}$
G ist kontextfrei und **nicht** regulär.
$L(G)$ ist regulär, denn $L(G)$ wird erzeugt von $S \to cX, X \to cX, X \to dY, Y \to dY, Y \to \varepsilon$ und diese Grammatik ist regulär!

Typ1-Sprachen heißen auch *kontextsensitiv*
Typ2-Sprachen heißen auch *kontextfrei*
Typ3-Sprachen heißen auch *regulär*

Definition. Man fasst alle Sprachen vom Typ i zu einer Familie $\mathcal{L}_i$ zusammen. Nach Definition der Grammatik vom Typ $i (i = 0, 1, 2, 3)$ folgt

$$\begin{array}{rcll} \mathcal{L}_i & \subseteq & \mathcal{L}_0 & (i = 1, 2, 3) \\ \mathcal{L}_3 & \subseteq & \mathcal{L}_2 & \end{array}$$

Es lässt sich zeigen: $\mathcal{L}_3 \subsetneqq \mathcal{L}_2 \subsetneqq \mathcal{L}_1 \subsetneqq \mathcal{L}_0$



Definition. Eine Sprachfamilie $\mathcal{L}$ heißt abgeschlossen gegenüber einer Operation, wenn die Operation angewandt auf Sprachen aus $\mathcal{L}$ wieder eine Sprache aus $\mathcal{L}$ liefert.

a) mengentheoretische Operationen:

$$\begin{array}{lcll} L_1 & \cup & L_2 & \text{Vereinigung} \\ L_1 & \cap & L_2 & \text{Durchschnitt} \\ L_1 & \setminus & L_2 & \text{Differenz} \\ \overline{L} & = & T^* \setminus L & \end{array}$$

b) Konkatenation (Verkettung)

$$L_1 L_2 = \{w_1 w_2 : w_1 \in L_1, w_2 \in L_2\}$$

$$\text{speziell } L^i := LL^{(i-1)}, L^0 = \{\varepsilon\} \text{ rekursiv}$$

c) Iteration

$$L^* = \prod_{i=0}^{\infty} L^i$$

Definition. Vereinigung, Konkatenation und Iteration heißen *reguläre Operationen*. Es gilt, dass $\mathcal{L}_i (i = 0, 1, 2, 3)$ abgeschlossen ist gegenüber regulären Operationen.

1.4 Reguläre Sprachen

Satz 1.5. (Hauptsatz für reguläre Sprachen) Für eine Sprache L sind folgende Bediningungen äquivalent:

(i) L wird von einem endlichen Automaten akzeptiert

(ii) $L \in \mathcal{L}_3$

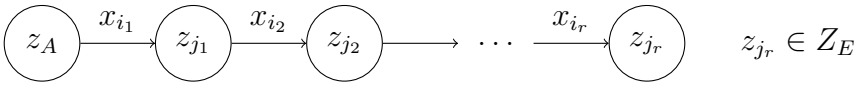
Beweis. $(i) \Longrightarrow (ii)$: Sei $A = (X, Z, f, z_A, Z_E)$ ein Akzeptor für $L \subseteq X^*$.
Wir konstruieren eine Grammatik $G = (N, T, S, P)$ wie folgt:

$$N := Z,\ T := X,\ S := z_A,\ P := \{z \to xz', \text{ wenn } f(z, x) = z'\} \cup \{z \to \varepsilon : z \in Z_E\}$$

(also: wenn $z \xrightarrow{x} z' \in A$, so $z \to xz'$ in P; wenn $\textcircled{z}$ in A, so $z \to \varepsilon$ in p) □

Behauptung. $L(G) = L(A)$

a) $w = x_{i_1} \dots x_{i_r} \in L(A)$, d.h.



Dann $z_A \Longrightarrow x_{i_1} z_{j_1} \Longrightarrow x_{i_1} x_{1_2} z_{j_2} \Longrightarrow \dots \Longrightarrow x_{i_1} \dots x_{i_r} z_{j_r} \Longrightarrow x_{i_1} \dots x_{i_r}$

b) $w \in L(G)$, also $z_A \overset{*}{\Longrightarrow} w$:
Wegen der Form der Grammatik gibt es dann nur die Ableitungsmöglichkeit

$$z_A \Longrightarrow x_{i_1} z_{j_r} \Longrightarrow \dots \Longrightarrow x_{i_1} \dots x_{i_r} z_{j_r} \Longrightarrow x_{i_1} \dots x_{i_r}$$

Nach Definition der Grammatik gilt gerade $f^*(z_A, w) = z_{j_r} \in Z_E$.

Beweis. $(ii) \Longrightarrow (i)$: Sei $G = (N, T, S, P)$ eine reguläre Grammatik mit $L = L(G)$.
Wir konstruieren eine zu G äquivalente Grammatik $G' = (N', T, S, P')$, deren Produktion von einer der folgenden zwei Formen ist:

(1) $Y \longrightarrow aX;\ a \in T;\ X, Y \in N'$

(2) $Y \longrightarrow \varepsilon;\ Y \in N'$

Außer Produktionen wie in (1) und (2) kann G Produktionen der folgenden drei Formen enthalten:

(3) $X \longrightarrow a_1 \dots a_r Y;\ r \geq 2;\ a_i \in T;\ X, Y \in N$

(4) $X \longrightarrow a_1 \dots a_r;\ r \geq 1;\ a_i \in T;\ X \in N$

(5) $X' \longrightarrow Y;\ X', Y \in N$

Elimination von Produktionen der Form (3):
Einführung **neuer** Nichtterminale $X_1, \dots, X_{r-1}$ und Ersetzen von (3) durch

$$X \longrightarrow a_1 X_1, X_1 \longrightarrow a_2 X_2, \dots, X_{r-1} \longrightarrow a_r Y$$

Hierbei ändert sich L offensichtlich nicht.
Dies fortsetzen, bis alle Regeln der Form (3) eliminiert sind.
Eliminieren von Produktionen der Form (4):
Neue Nichtterminale $Y_1, \dots, Y_r$ und Ersetzen von (4) durch

$$X \longrightarrow a_1 Y_1, Y_1 \longrightarrow a_2 Y_2, \dots, Y_{r-1} \longrightarrow a_r Y_r, Y_r \longrightarrow \varepsilon$$

Auch hierbei ändert sich L nicht!
Dies fortsetzen, bis alle Regeln der Form (4) eliminiert sind.
Dies liefert Grammatik $\overline{G}$; $\overline{G}$ enthält nur Produktionen der Form (1), (2) und (5) und $L = L(\overline{G})$

$$\begin{array}{ccccccccccc} X & \Longrightarrow & X_1 & \Longrightarrow & X_2 & \Longrightarrow & \dots & \Longrightarrow & X_r & \Longrightarrow & aX' \\ & & & & \Downarrow & & & & & & \\ & & & & X \Rightarrow aX' \text{ wenige } X \to aX' & & & & & & X \in U(X_r) \end{array}$$

Elimination von Produktionen der Form (5):
Für jedes $Z \in \overline{N}$ sei $U(Z) = \{Z' \in \overline{N} : Z' \stackrel{*}{\Longrightarrow} Z\}$.
Der fragliche Algorithmus liefert $U(Z)$:

$$
\begin{array}{rcl}
U(Z) & := & \{Z\} \\
\texttt{repeat} & & \\
U_{alt} & := & U(Z) \\
U(Z) & := & U_{alt} \cup \{X \in \overline{N} : X \longrightarrow Y \in \overline{P} : Y \in U_{alt}\} \\
\texttt{until} & & \\
U_{alt} & = & U(Z)
\end{array}
$$

Sei jetzt $N' := \overline{N}$, P' werde aus $\overline{P}$ wie folgt gebildet:

- Alle Produktionen der Form $X' \longrightarrow Y$ werden gestrichen
- Ist $Y \longrightarrow aX \in \overline{P}$, so wird für alle $Y' \in U(Y)$ die Produktion $Y' \longrightarrow aX$ in P' hinzugenommen
- Ist $Y \longrightarrow \varepsilon \in \overline{P}$, so wird für alle $Y' \in U(Y)$ die Produktion $Y' \longrightarrow \varepsilon$ in P' hinzugenommen.

G' hat die verlangte Form. Man kann zeigen, dass $L(\overline{G}) = L(G')$ ist.
Wir können also annehmen, dass nur Produktionen der Form (1) und (2) in G' vorkommen.
Dann werde

$$A = (T, N', R_f, S, \{X \in N' : X \longrightarrow \varepsilon \in P'\})$$

definiert mit

$$(Y, a, X) \in R_f \text{ g.d.w. } Y \longrightarrow aX \in P'$$

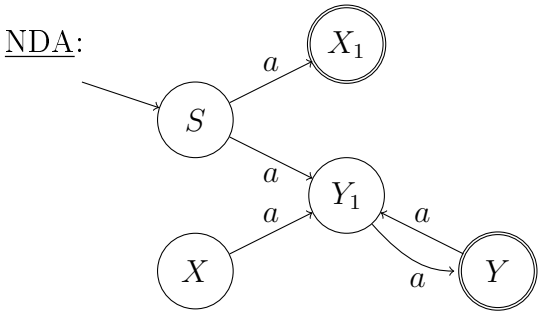
Offenbar gibt es einen Weg von S vermöge $w \in T^*$ in einen Zustand aus Z_E g.d.w. $S \stackrel{*}{\Longrightarrow} w$ □

Beispiel. $G = (\{S, X, Y\}, \{a, b\}, S, P)$
mit $P = \{S \to a, S \to X, X \to abY, Y \to X, Y \to \varepsilon\}$
Dann $\overline{G}$ mit $\overline{N} = \{S, X, Y, X_1, Y_1\}$ und

$$\overline{P} = \{S \to aX_1, X_1 \to \varepsilon, S \to X, X \to aY_1, Y_1 \to bY, Y \to X, Y \to \varepsilon\}$$

Man erhält $U(X) = \{X, S, Y\}$, $U(S) = \{S\}$, ...
Also $S \to X$ und $Y \to X$ streichen und neben $X \to aY_1$ noch $S \to aY_1$ und $Y \to aY_1$ in P', also

$$P' = \{S \to aX_1, X_1 \to \varepsilon, X \to aY_1, S \to aY_1, Y \to aY_1, Y_1 \to bY, Y \to \varepsilon\}$$



Knoten X und zugehörige Kante können gestrichen werden.

Folgerung. $\mathcal{L}_3$ ist abgeschlossen gegenüber Vereinigung, Durchschnitt und Komplement.

Beweis.

a) Seien L_1 und L_2 regulär. Nach Satz gibt es DAs A_1 bzw. A_2, die L_1 bzw. L_2 akzeptieren. Die „*disjunkte Vereinigung*“ von A_1 und A_2 ist ein NDA, der genau $L_1 \cup L_2$ akzeptiert.

b) Sei $A_1 = (X, Z, f, z_A, Z_E)$; dann $\overline{A}_1 = (X, Z, f, z_A, Z \setminus Z_E)$ und $L(\overline{A}_1) = \overline{L}_1$.

c) $L_1 \cap L_2 = \overline{L}_1 \cup \overline{L}_2$

□

Bemerkung. Bei der Definition der regulären Sprachen wurden *rechts*-reguläre Grammatiken benutzt. Man kann *links*-reguläre Grammatiken analog definieren als solche mit Produktionen der Form $X \to Yw$ und $X \to w$ $(X, Y \in N, w \in T^*)$.
Das liefert nichts Neues, denn

Satz 1.6. Die Klasse $\mathcal{L}_3$ stimmt mit der Klasse der links-regulären Sprachen überein.

Beachte. Es dürfen **nicht** sowohl Produktionen der Form $X \to Yw$ und $X \to wY$ vorkommen!

Bemerkung. Der Nachweis, dass die Sprache L regulär ist, kann durch die Angabe eines Akzeptors für L oder durch Angabe einer erzeugenden regulären Grammatik für L erfolgen.

Frage. Wie kann man zeigen, dass eine Sprache nicht regulär ist?

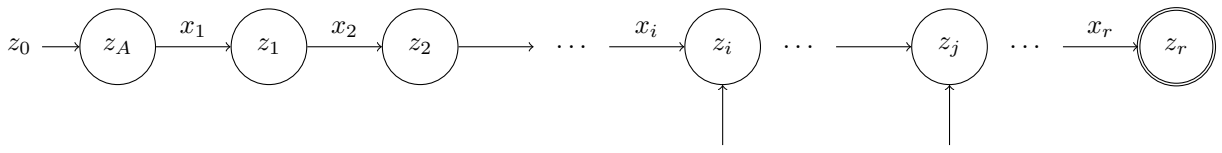
Hierzu benutzt man (häufig) folgendes Resultat:

Satz 1.7 (*Pumping Lemma* für reguläre Sprachen)**.** Sei L eine reguläre Sprache. Dann existiert ein $n > 0$ (abhängig von L), so dass $\forall p \in L$ mit $|p| \geq n$ gilt:
Es gibt u, v, w mit $p = uvw$ und $|uv| \leq n$, $|v| \geq 1$ und $uv^i w \in L \; \forall i \geq 0$.

$$v^i = \underbrace{vv \ldots v}_{i\text{-mal}} \qquad L = \{a^i b^i : i \geq 1\}$$

(p verbleibt - „*beliebig aufgepumpt*“ - in L)

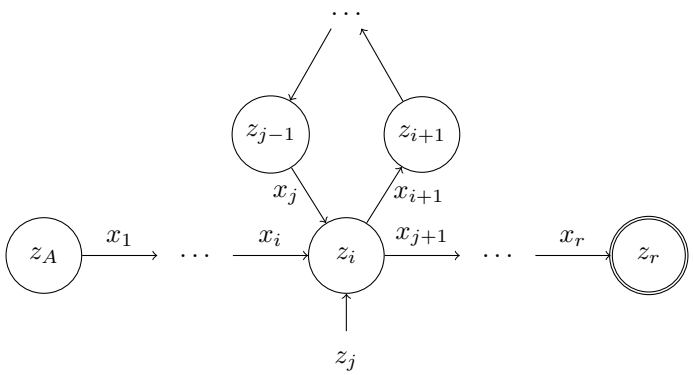
Beweis. Es gibt für L einen Akzeptor $A = (X, Z, f, z_A, Z_E)$.
Sei $n := |Z|$ und $p \in L, |p| \geq n, p = x_1 \ldots x_r (r \geq n)$.



$z_r \in Z_E$

Wegen $r \geq n$ sind zwei der durchlaufenen Zustände identisch; wir wählen die ersten beiden solchen (2. Index möglichst klein) etwa z_i und z_j.

$$u := x_1 \ldots x_i, v := x_{i+1} \ldots x_j, w = x_{j+1} \ldots x_r$$



Es ist $p = uvw, |uv| \leq n$ wegen der *Wahl* von z_i und z_j, $|v| \geq 1$ nach Konstruktion und $uv^iw \in L\ \forall i \geq 0$, weil der Zyklus beliebig oft durchlaufen werden kann. □

Beispiel. Sei $L = \{a^jb^j : j \in \mathbb{N}\}$. Wäre L regulär, so existiert $n > 0$ mit ...
Wir wählen $p = a^nb^n \in L$, wegen $|uv| \leq n \curvearrowright uv = a^k, v = a^l$ mit $l \geq 1$. Dann lägen alle $a^{k-l}(a^l)^ia^{n-k}b^n$ $(i = 0, 1, 2, \ldots)$ in L. (z.B. $i = 0$ oder $i = 2$) ↯

Bemerkung. Man kann a als öffnende und b als schließende Klammern interpretieren. Dann ist L die Menge der einfachsten wohlgeformten Klammerstrukturen beliebiger Tiefe! (z.B. Klammerung von Blöcken in PASCAL mit `begin` und `end`)
Man kann also solche Sprachen **nicht** mit endlichen Automaten analysieren.
$\Longrightarrow$ Reguläre Sprachen sind zu eingeschränkt (für die Analyse von Programmiersprachen).

1.5 Kontextfreie Sprachen

BNF-Systeme sind kontextfreie Grammatiken!
Kontextfreie Grammatiken und Sprachen sind also wichtig für Theorie der Programmiersprachen.

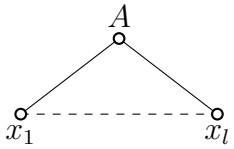
Ableitungsbäume
Ableitungen in kontextfreien Grammatiken werden häufig *Ableitungsbäume* zugeordnet.
Sei $G = (N, T, S, P)$ eine kontextfreie Grammatik.
Jeder Ableitung

$$S \Longrightarrow w_1 \Longrightarrow w_2 \Longrightarrow \ldots \Longrightarrow w_{n-1} \Longrightarrow w_n$$

wird ein Ableitungsbaum mit der Wurzel S wie folgt zugeordnet:

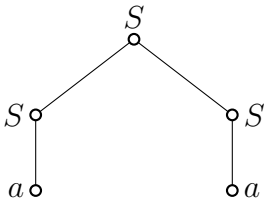
- Für $n = 0$ besteht der Baum nur aus dem Knoten S
- Ist B_{n-1} der Ableitungsbaum von $S \Longrightarrow w_1 \Longrightarrow \ldots \Longrightarrow w_{n-1}$ und entsteht w_n aus w_{n-1} durch Anwendung der Regel $A \longrightarrow w = x_1 \ldots x_l,\ x_i \in N \cup T,\ A \in N$,
  d.h. $w_{n-1} = w'Aw'', w_n = w'ww''$

so erhält man den Ableitungsbaum B_n von $S \Longrightarrow w_1 \Longrightarrow \ldots \Longrightarrow w_{n-1} \Longrightarrow w_n$ aus B_{n-1}, indem der Knoten A in B_{n-1} durch den Baum



ersetzt wird.

Beispiel. Sei $G = (\{S\}, \{a\}, S, \{S \to a, S \to SS\})$.
Die Ableitung $S \Longrightarrow SS \Longrightarrow aS \Longrightarrow aa$ hat den Ableitungsbaum



Offenbar können verschiedene Ableitungen den gleichen Ableitungsbaum haben, z.B. hat die Ableitung

$$S \Longrightarrow SS \Longrightarrow Sa \Longrightarrow aa$$

den gleichen Ableitungsbaum!
Die Zuordnung von Ableitungsbäumen zu Ableitungen wird eineindeutig, wenn man nur *Linksableitungen* zulässt.

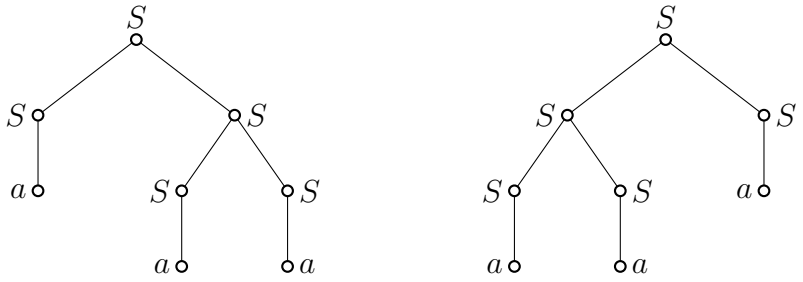
Definition (*Linksableitung*)**.** Die Ableitung $S \Longrightarrow w_1 \Longrightarrow \ldots \Longrightarrow w_n$ heißt Linksableitung, wenn in jedem Ableitungsschritt das am Weitesten links stehende Nichtterminal ersetzt wird (nach einer Regel aus P).

Es ist aber möglich, dass es für ein Wort $w \in L(G)$ verschiedene Linksableitungen gibt:

Definition (*mehrdeutige* Grammatik)**.** Die kontextfreie Grammatik G heißt mehrdeutig, wenn $\exists w \in L(G)$, so dass w zwei verschiedene Linksableitungen besitzt.
Andernfalls heißt G eindeutig.
Die kontextfreie Sprache L heißt eindeutig, wenn es eine eindeutige kontextfreie Grammatik G gibt mit $L = L(G)$.
Anderenfalls heißt L mehrdeutig.

Beispiel. $G = (\{S\}, \{a\}, S, \{S \to a, S \to SS\})$ ist mehrdeutig, denn $aaa \in L(G)$ hat zwei Linksableitungen

$$S \Longrightarrow SS \Longrightarrow aS \Longrightarrow aSS \Longrightarrow aaS \Longrightarrow aaa$$
$$S \Longrightarrow SS \Longrightarrow SSS \Longrightarrow aSS \Longrightarrow aaS \Longrightarrow aaa$$



Es ist $L(G) = \{a^i : i \geq 1\}$. $L(G)$ ist (natürlich) regulär, z.B. erzeugt von der Grammatik $G_1 = (\{S\}, \{a\}, S, \{S \to a, S \to aS\})$.
G_1 ist offenbar eindeutig, also auch $L(G_1) = L(G)$.

Beispiel (für eine mehrdeutige Sprache)**.**

$$L = \{a^i b^j c^k : i = j \text{ oder } j = k\} \quad \text{o.B.}$$

Dagegen gilt:

Satz 1.8. Jede reguläre Sprache ist eindeutig.

Beweis. Sei L regulär. Dann wird (nach Haupsatz) L von einem DA akzeptiert. Im Beweisteil (i) $\Longrightarrow$ (ii) wird dann gerade eine eindeutige (reguläre) Grammatik G mit $L = L(G)$ konstruiert. □

Normalformen
Es gibt für eine kontextfreie Sprache L viele verschiedene Grammatiken, die L erzeugen. Für Einsichten in die Struktur der kontextfreien Sprachen sind gewisse „*Normierungen*" für erzeugende Grammatiken wichtig. Hier sei nur eine erwähnt:

Definition. Eine kontextfreie Grammatik $G = (N, T, S, P)$ ist CHOMSKY-Normalform, wenn alle Regeln von der Form sind $X \longrightarrow YZ$ oder $X \longrightarrow a$ mit $X, Y, Z \in N$ und $a \in T$ sind.

Satz 1.9. Zu jeder kontextfreien Grammatik $G = (N, T, S, P)$ mit $\varepsilon \notin L(G)$ gibt es eine äquivalente kontextfreie Grammatik G' in CHOMSKY-Normalform. (o.B.)

Zum *Pumping-Lemma* für reguläre Sprachen gibt es ein Analogon:

Satz 1.10 (*Pumping-Lemma* für kontextfreie Sprachen)**.** Sei L kontextfrei. Dann gibt es eine natürliche Zahl $n > 0$, so dass für jedes Wort $w \in L$ mit $|w| \geq n$ gilt:
Es gibt w_1, w_2, w_3, w_4, w_5 mit
$w = w_1w_2w_3w_4w_5$, $|w_2w_3w_4| \leq n$, $w_2w_4 \neq \varepsilon$, $w_1w_2^iw_3w_4^iw_5 \in L\ \forall i \geq 0$ (o.B.)

Da Pumping-Lemma erlaubt häufig den Nachweis, dass eine Sprache nicht kontextfrei ist.

Beispiel. $L = \{vv : v \in \{0,1\}^*\}$
Angenommen, L wäre kontextfrei. Dann existiert n mit Eigeschaften des Pumping-Lemmas.
Wir wählen:

$$w = \underbrace{1\ldots1}_{n}\underbrace{0\ldots0}_{n}\underbrace{1\ldots1}_{n}\underbrace{0\ldots0}_{n} \in L$$

Dann wäre also $w = w_1\underline{w_2w_3w_4}w_5$ mit $|w_2w_3w_4| \leq n$
Wenn $w_2w_3w_4$ in der 1. Hälfte von w liegt, so endet im für $i = 0$ erzeugten Wort die 1. Hälfte mit 1, die 2. mit 0 (sofern überhaupt ein Wort gerader Länge entsteht).
Wenn $w_2w_3w_4$ in der 2. Hälfte von w liegt, so beginnt im für $i = 0$ erzeugten Wort die 1. Hälfte mit 1, die 2. mit 0 (sofern überhaupt ein Wort gerader Länge entsteht).
Wenn $w_2w_3w_4$ die Mitte überlappt, so beginnt im für $i = 0$ erzeugten Wort die 1. Hälfte mit n Einsen die 2. Hälfte nicht. ↯

Bemerkung. Dieses Beispiel dokumentiert die mangelnde „*Kopierfähigkeit*" kontextfreier Grammatiken. Eine Konsequenz ist, dass die meisten Programmiersprachen nicht kontextfrei sind.
Z.B. gibt es in PASCAL (und C) die Bedingung, dass Anzahl und Typen der formalen Parameter und der aktuellen Parameter übereinstimmen müssen.
Diese Bedingung lässt sich - unserem obigen Beispiel folgend - **nicht** durch BNF-Regeln formulieren! Bei Programmiersprachen erzeugt man durch kontextfreie Grammatiken (BNF-Regeln) eine echte Obermenge der zulässigen Programme, aus denen die zulässigen durch verbale Einschränkungen ausgesondert werden.

Satz 1.11. Kontextfreie Sprachen sind abgeschlossen gegenüber den regulären Operationen Vereinigung, Verkettung und Iteration.

Beweis. Seien $L_1 = L(G_1)$ und $L_2 = L(G_2)$ erzeugt von den kontextfreien Grammatiken $G_1 = (N_1, T_1, S_1, P_1)$ bzw. $G_2 = (N_2, T_2, S_2, P_2)$ und o.B.d.A. $N_1 \cap N_2 = \varnothing$.
Seien S_3, S_4, S_5 neue Nichtterminale.

$$G_3 := (N_1 \cup N_2 \cup \{S_3\}, T_1 \cup T_2, S_3, P_3)$$

mit $P_3 = P_1 \cup P_2 \cup \{S_3 \longrightarrow S_1, S_3 \longrightarrow S_2\}$
Offenbar ist $L(G_3) = L_1 \cup L_2$.
Randnotiz: $L_1L_2 = \{w_1w_2 : w_1 \in L_1, w_2 \in L_2\}$

$$G_4 := (N_1 \cup N_2 \cup \{S_4\}, T_1 \cup T_2, S_4, P_4)$$

mit $P_4 = P_1 \cup P_2 \cup \{S_4 \longrightarrow S_1 S_2\}$
Offenbar ist $L(G_4) = L_1 L_2$.

$$G_5 := (N_1 \cup \{S_5\}, T_1 \cup T_2, S_5, P_5)$$

mit $P_5 = P_1 \cup \{S_5 \longrightarrow \varepsilon, S_5 \longrightarrow S_1 S_5\}$
also

$$L_1^* = \bigcup_{i=0}^{\infty} L_1^i$$

Also ist $L(G_5) = L_1^*$. □

Satz 1.12. Die kontextfreien Sprachen sind nicht abgeschlossen gegenüber Durchschnitt.

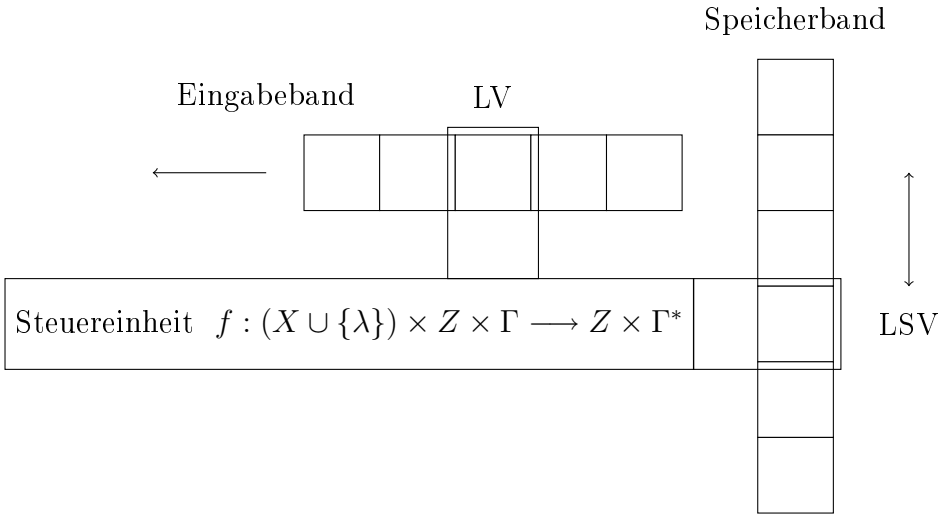
Beweis.

$$L_2 := \{a^i b^i c^j : i \geq 1, j \geq 1\}$$
$$L_3 := \{a^i b^j c^j : i \geq 1, j \geq 1\}$$

L_2 wird erzeugt von der Grammatik mit den Produktionen $S \to AB, A \to aAb \mid ab, B \longrightarrow cB \mid c$
L_3 wird erzeugt von der Grammatik mit den Produktionen $S \to CD, C \to aC \mid a, D \longrightarrow bDc \mid bc$
Aber $L_2 \cap L_3 = \{a^i b^i c^i : i \geq 1\}$ ist nicht kontextfrei. □

1.6 Kellerautomaten

Erweiterung des Automatenmodells um einen Speicher in der folgenden Weise:



$X = \{x_1, \ldots, x_n\}$ Eingabealphabet

$Z = \{z_1, \ldots, z_k\}$ Menge von Zuständen

$\Gamma = \{K_1, \ldots, K_m\}$ Kelleralphabet

Ferner sei λ (Leerzeichen) $\notin X$.

$$f : (X \cup \{\lambda\}) \times Z \times \Gamma \longrightarrow Z \times \Gamma^* \quad \textit{partielle} \text{ Funktion}$$

Dabei:

- Eingabeband kann in jedem Feld jeweils ein Eingabezeichen enthalten
- LV gestattet Einlesen von jeweils einem Eingabezeichen
- Jedes Feld des Speicherbandes enthält höchstens ein Kellerzeichen
- LSV gestattet Lesen von jeweils einem Kellerzeichen (dem „*obersten*“, über dem stehts die LSV steht) und Schreiben eines Wortes aus Γ^* ab dem Feld (einschließlich), über dem die LSV steht (nach oben)
- Speicherband kann in beide Richtungen bewegt werden
- Speichern von ε (leeres Wort) auf Speicherband bedeutet Löschen des dort stehenden Zeichens

$z_A \in Z$ Startzustand

$K_A \in \Gamma$ Kellerstartzeichen

$Z_E \subseteq Z$ Menge von Endzuständen

Definition. Ein deterministischer Kellerautomat A ist ein 7-Tupel $A = (X, Z, \Gamma, z_A, K_A, f, Z_E)$ mit endlichen nichtleeren Mengen $X, Z, \Gamma, z_A \in Z, Z_E \subseteq Z, K_A \in \Gamma, \lambda \notin X$ und $f : (X \cup \{\lambda\}) \times Z \times \Gamma \longrightarrow Z \times \Gamma^*$ *partiell*, wobei f die folgende Bedingung erfüllt:
Wenn $f(\lambda, z, K)$ definiert ist, so ist $\forall x \in X$ $f(x, z, K)$ nicht definiert.
Der Gesamtzustand eines KA ist beschrieben durch ein Tripel

$$(w, z, \alpha), w \in X^*, z \in Z, \alpha \in \Gamma^*$$

Ein solches Tripel heißt *Konfiguration*.
Eine Konfiguration kann interpretiert werden:

- w ist ein noch nicht gelesenes Endstück eines Eingabewortes
- z ist der Zustand
- α ist der Zustand des Kellerbandes

f definiert nun gerde eine Übertragungsrelation $\vdash$ zwischen Konfigurationen:

$$\begin{aligned} (w, z, K\alpha) &\vdash (w, z', \beta\alpha) \text{ , wenn } f(\lambda, z, K) = (z', \beta) \\ (xw, z, K\alpha) &\vdash (w, z', \beta\alpha) \text{ , wenn } f(x, z, K) = (z', \beta) \end{aligned}$$

$(x \in X, K \in \Gamma)$
Verbale Beschreibung des Verhaltens:

(a) Wenn der Keller leer ist, so bleibt KA stehen (kein Übergang möglich)

(b) Wenn Zustand z und oberstes Kellerzeichen K und $f(\lambda, z, K)$ definiert ist und gleich (z', β), so geht KA in Zustand z' über, K wird ersetzt durch β und Eingabeband bleibt stehen (von Eingabeband wird nicht gelesen)

(c) Wenn Konfiguration $(xw, z, K\alpha)$ und $f(x, z, K) = (z', \beta)$, so geht KA in Zustand z' über, K wird durch β ersetzt und Eingabeband bewegt sich um ein Feld nach links

(d) Wenn weder (b) noch (c) zutrifft, so bleibt KA stehen (kein Übergang möglich)

Definition. $\overset{*}{\vdash}$ sei die reflexive transitive Hülle von $\vdash$.
$w \in X^*$ wird vom KA A akzeptiert, wenn gilt

$$(w, z_A, K_A) \overset{*}{\vdash} (\varepsilon, z_E, \alpha) \text{ für ein } z_E \in Z_E, \alpha \in \Gamma^*$$

$$L(A) := \{w \in X^* : w \text{ wird von } A \text{ akzeptiert } \} \text{ ist die Sprache von } A$$

Beispiel. $A = (\{a, b\}, \{z_1, z_2, z_3\}, \{K, A\}, z_1, K, f, \{z_3\})$

$$\begin{aligned} f: \ (a, z_1, K) &\longrightarrow (z_1, AK) \\ (a, z_1, A) &\longrightarrow (z_1, AA) \\ (b, z_1, A) &\longrightarrow (z_2, \varepsilon) \\ (b, z_2, A) &\longrightarrow (z_2, \varepsilon) \\ (\lambda, z_2, K) &\longrightarrow (z_3, \varepsilon) \end{aligned}$$

$L(A) = \{a^i b^i : i \geq 1\}$; z.B. $w = a^2b^2$

Eingabeband	Zustand	Keller	Konfiguration
$\ldots \lambda \overset{\downarrow}{a}\, abb\lambda \ldots$	z_1	$\ldots \lambda\lambda \overset{\downarrow}{K} \mid$	$(aabb, z_1, K)$
$\ldots \lambda a \overset{\downarrow}{a}\, bb\lambda \ldots$	z_1	$\ldots \lambda \overset{\downarrow}{A}\, K \mid$	(abb, z_1, AK)
$\ldots \lambda aa \overset{\downarrow}{b}\, b\lambda \ldots$	z_1	$\ldots \lambda \overset{\downarrow}{A}\, AK \mid$	(bb, z_1, AAK)
$\ldots \lambda aab \overset{\downarrow}{b}\, \lambda \ldots$	z_2	$\ldots \lambda \overset{\downarrow}{A}\, K \mid$	(b, z_2, AK)
$\ldots \lambda aabb \overset{\downarrow}{\lambda} \ldots$	z_2	$\ldots \lambda \overset{\downarrow}{K} \mid$	(ε, z_2, K)
$\ldots \lambda aabb \overset{\downarrow}{\lambda} \ldots$	z_3	$\ldots \lambda \mid$	$(\varepsilon, z_3, \varepsilon)$

also $a^2b^2 \in L(A)$.
Offenbar wird für jedes $i \geq 1$

$$(a^i b^i, z_1, K) \vdash (a^{i-1} b^i, z_1, AK) \overset{*}{\vdash} (b^i, z_1, A^i K) \vdash (b^{i-1}, z_2, A^{i-1} K) \overset{*}{\vdash} (\varepsilon, z_2, K) \vdash (\varepsilon, z_3, \varepsilon)$$

Also $\{a^i b^i : i \geq 1\} \subseteq L(A)$.
Sei $w \in L(A) \Longrightarrow w \neq \varepsilon$ (da $f(\lambda, z_1, K)$ fehlt). w kann nicht mit b beginnen, da $f(b, z_1, K)$ fehlt. Wenn auf gewisse a's ein b folgt, so Übergang zu z_2. Dann kann kein a mehr folgen, weil $f(a, z_2, \ldots)$ fehlt. Folgt auf a^i nur b^j, $j < i$, so fehlt $f(\lambda, z_2, A)$. Nach $a^i b^i$ ist Konfiguration $(\ldots, z_2, K)$ und hierauf kann kein b mehr folgen. Also $L(A) \subseteq \{a^i b^i : i \geq 1\}$.

Bemerkung. Wenn man $\varepsilon \in L$ haben will, das heißt $L = \{a^i b^i : i \geq 0\}$, somit zusätzlicher Startzustand z_0 (z_1 dann kein Startzustand):

$$\begin{aligned} (\lambda, z_0, K) &\longrightarrow (z_3, K) \\ (a, z_3, K) &\longrightarrow (z_1, AK) \\ &\vdots \end{aligned}$$

Wir betrachten ein weiteres

Beispiel. Ein *Palindrom* ist ein Wort, das gleich seiner Umkehrung ist.
Sei w^R die Umkehrung von w.

$$L := \left\{ww^R : w \in \{a, b\}^*\right\}$$

L ist kontextfrei; man überzeuge sich, dass L von der Grammatik mit den Produktionen $S \longrightarrow aSa, S \longrightarrow bSb, S \longrightarrow \varepsilon$ erzeugt wird.
Man kann zeigen, dass es keinen deterministischen KA gibt, der genau L akzeptiert.
(Intuitive Erklärung: ein KA kann zwar den Anfang des Wortes im Keller ablegen, *weiß* aber nicht, wann die Mitte erreicht ist und er also mit dem Vergleich anfangen soll)

Definition. Ein nichtdeterministischer Kellerautomat (NKA) ist ein 7-Tupel
$A = (X, Z, \Gamma, z_A, K_A, R_f, Z_E)$ mit $X, Z, \Gamma, z_A, K_A, Z_E$ wie oben und $R_f \subseteq (X \cup \{\lambda\}) \times Z \times \Gamma \times Z \times \Gamma^*$ sei eine *endliche* Relation.
Eine Konfiguration ist erklärt wie oben und $\vdash$ analog:

$$\begin{aligned} (w, z, K\alpha) &\vdash (w, z', \beta\alpha) \text{ , wenn } (\lambda, z, K, z', \beta) \in R_f \\ (xw, z, K\alpha) &\vdash (w, z', \beta\alpha) \text{ , wenn } (x, z, K, z', \beta) \in R_f \end{aligned}$$

Ein vom NKA akzeptiertes Wort und die vom NKA akzeptierte Sprache sind wie oben erklärt.

f ersetzen durch Relation R_f (endlich).
Sei $A = (\{a, b\}, \{z_1, z_2, z_3\}, \{K, A, B\}, z_1, K, R_f, \{z_3\})$ mit

$R_f = \{$

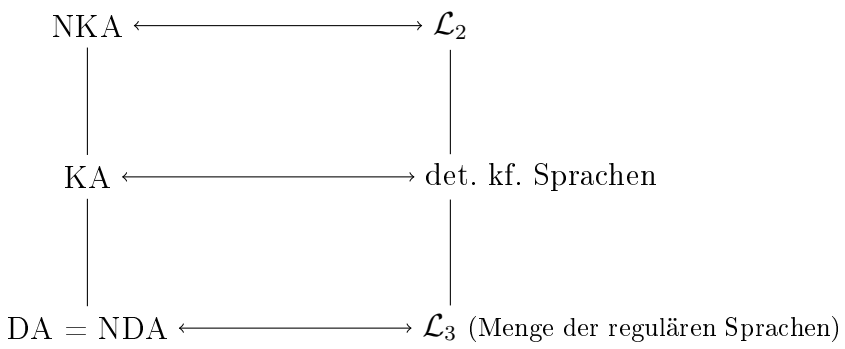
$(\lambda, z_1, K, z_3, \varepsilon)$	leeres Wort wird akzeptiert
(a, z_1, K, z_1, AK), (b, z_1, K, z_1, BK)	1. Zeichen abkellern
(a, z_1, B, z_1, AB), (b, z_1, A, z_1, BA)	nächstes Zeichen abkellern, wenn Vorgänger davon verschieden
(a, z_1, A, z_1, AA), $(a, z_1, A, z_2, \varepsilon)$	nächstes Zeichen abkellern oder Versuch umzukehren
(b, z_1, B, z_1, BB), $(b, z_1, B, z_2, \varepsilon)$	nächstes Zeichen abkellern oder Versuch umzukehren
$(a, z_2, A, z_2, \varepsilon)$, $(b, z_2, B, z_2, \varepsilon)$	Überprüfen, ob Palindrom
$(\lambda, z_2, K, z_3, \varepsilon)$	$\}$

Man kann sich exakt überlegen, dass $L = L(A)$.
Nun zeigt sich gerade, dass die so definierten NKA gerade zu den kontextfreien Sprachen „*passen*“:

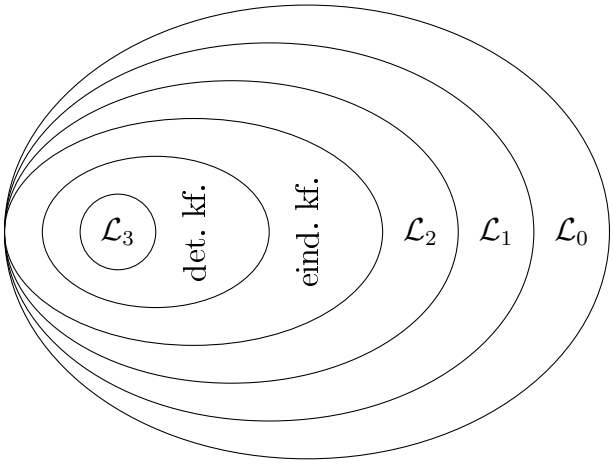
Satz 1.13. Zu jeder kontextfreien Sprache L existiert ein (i.A.) nichtdeterministischer KA M derart, dass

$$L = L(M)$$

Umgekehrt gibt es zu jedem NKA M eine kontextfreie Grammatik G mit $L(M) = L(G)$. (o.B.)



Und für die Sprachfamilien insgesamt



Wortproblem für kontextfreie Sprachen:
Gegeben sei eine kontextfreie Grammatik $G = (N, T, S, P)$ und $w \in T^*$.
Liegt w in $L(G)$ und wenn ja, was ist ein Ableitungsbaum für w in G?

Wichtig für Syntaxanalyse von Programmiersprachen!

Zur Lösung des Wortproblems sind Kellerautomaten nicht geeignet, da sie i.A. *nichtdeterministisch* sein müssen und die Lösung des Wortproblems dann zu aufwendig ist.

Satz 1.14. Es gibt einen Algorithmus (COCKE, YOUNGER, KASAMI), der das Wortproblem für kontextfreie Grammatiken in CHOMSKI-Normalform in $O(n^3)$ löst (wobei $n = |w|$). (o.B.)

Bemerkung. Dies ist aber für die Syntaxanalyse von Programmiersprachen zu langsam.
Man beschränkt sich daher bei Programmiersprachen auf Teilklassen kontextfreier Sprachen, z.B. von sogenannten $LR(k)$-Grammatiken erzeugte Sprachen.
$\rightsquigarrow$ Compilerbau + Übersetzertechnik

Kapitel 2

Algorithmen, Entscheidbarkeit, Berechenbarkeit

2.1 TURING-Maschinen

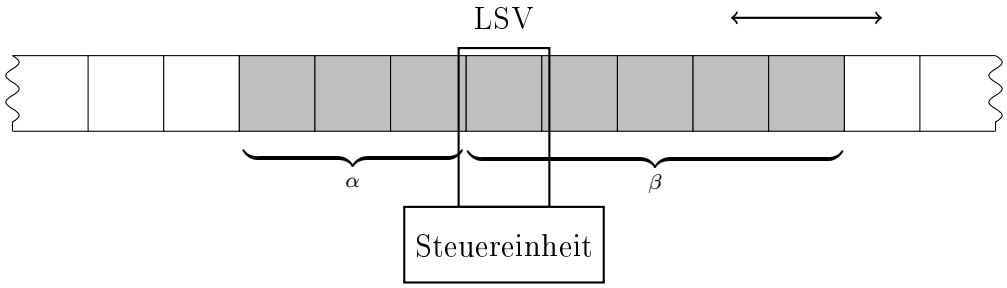
Algorithmus - zentraler Begriff in der Informatik. Übliche Definition des Begriffes nicht scharf; Probleme entstehen sofort, wenn die Frage steht, ob es zu einem vorgegebenen Problem überhaupt einen Algorithmus gibt. Um *beweisen* zu können, dass es einen solchen **nicht** gibt, braucht man eine exakte *Definition* des Begriffes Algorithmus!

2.1.1 Definition von TURING-Maschinen

Alan TURING (1912*- 1954†)
„On computable numbers with an application on the *Entscheidungsproblem*“ (1936)

Definition (*Entscheidungsproblem*)**.** Gegeben ist irgendeine *beliebige* mathematische Behauptung. Gibt es ein „*mechanisches Verfahren*“ das es zu entscheiden gestattet, ob diese Behauptung richtig oder falsch ist?

Modell:



- Steuereinheit besteht aus endlich vielen Zuständen
- Band auf beiden Seiten unbegrenzt
- Band ist in Zellen eingeteilt; in jeder Zelle steht ein Bandsymbol, wobei fast alle (d.h. bis auf endlich viele) Zellen das spezielle Bandsymbol λ (Leerzeichen) enthalten
- LSV kann den Inhalt einer Zelle lesen und verändern
- Ein Einzelschritt einer TM besteht (in Abhängigkeit vom Zustand und vom gelesenen Symbol) aus folgenden Aktionen

(1) Das gelesene Symbol wird gelöscht und an seine Stelle wird ein neues Symbol geschrieben

(2) Die LSV rückt um eine Zelle nach links oder rechts

(3) Die TM geht in einen neuen Zustand über

Bemerkung. Im Unterschied zu den anderen Automatenmodellen kann eine TM die Zeichen auf dem (Eingabe-) Band abändern (und später wieder lesen).

Definition. Eine (zweiseitige deterministische) TURING-Maschine

$$T = (\Sigma, Z, \Gamma, z_A, f, Z_E)$$

ist bestimmt durch

- eine endliche Menge von Eingabezeichen Σ
- eine endliche Menge von Zuständen Z
- eine endliche Menge von Bandsymbolen Γ; dabei enthält Γ das Leerzeichen λ und es ist $\Sigma \subseteq \Gamma \setminus \{\lambda\}$
- einen Anfangszustand $z_A \in Z$
- eine i.A. partielle Übergangsfunktion

$$f : Z \times \Gamma \longrightarrow Z \times \Gamma \times \{L, R\}$$

- eine Menge $Z_E \subseteq Z$ von Endzuständen.

Der Gesamtzustand einer TM T wird dargestellt durch ein Wort $\alpha z \beta \in \Gamma^* Z\ \Gamma^*$ und heißt Konfiguration.
(Annahme $\Gamma \cap Z = \varnothing$)
Das bedeutet: $\alpha\beta$ auf dem Band; Zustand z; LSV über 1. Zeichen von β.

Definition (*Berechnungsschritt*)**.** Ein Berechnungsschritt einer TM T wird durch das Symbol $\vdash$ dargestellt, genauer:

(1) $\alpha z_1 A \beta \vdash \alpha B z_2 \beta :\Longleftrightarrow f(z_1, A) = (z_2, B, R)$

(2) $\alpha C z_1 A \beta \vdash \alpha z_2 C B \beta :\Longleftrightarrow f(z_1, A) = (z_2, B, L)$

$\alpha z A \beta$ ist Endkonfiguration g.d.w. $z \in Z_E$ oder $f(z, A)$ nicht definiert.

Die TM T *hält* mit der Eingabe $w \in \Sigma^*$ g.d.w. $z_A w \overset{*}{\vdash} \alpha z \beta$ mit $\alpha z \beta$ Endkonfiguration.

Definition. Die von der TM T akzeptierte Sprache $L(T)$ ist definiert durch

$$L(T) := \{w \in \Sigma^*, z_A w \overset{*}{\vdash} \alpha z \beta, z \in Z_E\}$$

Beispiel. TM, die die Sprache $\{0^i 1^i, i \in \mathbb{N}, i \geq 1\}$ akzeptiert.

$$T = (\{0,1\}, \{z_1, \ldots, z_5\}, \{0, 1, X, Y, \lambda\}, z_1, f, \{z_5\})$$

f	0	1	X	Y	λ
z_1	(z_2, X, R)	–	–	(z_4, Y, R)	–
z_2	$(z_2, 0, R)$	(z_3, Y, L)	–	(z_2, Y, R)	–
z_3	$(z_3, 0, L)$	–	(z_1, X, R)	(z_3, Y, L)	–
z_4	–	–	–	(z_4, Y, R)	(z_5, Y, R)
z_5	–	–	–	–	–

TURING-Tafel

- in z_1 wird die am Weitesten links stehende 0 überschrieben mit X und in z_2 gegangen
- in z_2 werden die folgenden Nullen und Y's übergangen bis zur ersten 1, die dann durch Y ersetzt wird
- in z_3 werden nach links die Y's und Nullen übergangen bis zum ersten X, wo nach rechts gegangen und in z_1 gewechselt wird

Beispiel (Wort „0011“).

$$z_1 0011 \vdash X z_2 011 \vdash X 0 z_2 11 \vdash X z_3 0 Y 1 \vdash z_3 X 0 Y 1 \vdash X z_1 0 Y 1 \vdash X X z_2 Y 1$$

$$\vdash X X Y z_2 1 \vdash X X z_3 Y Y \vdash X z_3 X Y Y \vdash X X z_1 Y Y \vdash X X Y z_4 Y \vdash X X Y Y z_4 \vdash X X Y Y Y z_5$$

(a) $\{0^i 1^i, i \geq 1\} \subseteq L(T)$

$$X^j z_1 0^n Y^j 1^n \overset{*}{\vdash} \alpha z_5 \beta, \forall n > 0, j \geq 0$$

vollständige Induktion nach n

(b) $L(T) \subseteq L = \{0^i 1^i, i \geq 1\}$
Sei $w \in L(T) \Longrightarrow w$ beginnt mit 0

$\alpha)$ $w = 0^i 1^k w_1, 0 < k < i$, wenn $w_1 \neq \varepsilon$ so $w_1 = 0 w_2$

$\beta)$ $w = 0^i 1^i w_1$,

$\gamma)$ $w = 0^i 1^k w_1, k > i > 0$,

gälte α): $z_1 w \overset{*}{\vdash} X^k z_1 0^{i-k} Y^k w_1 \overset{*}{\vdash} X^{k+1} 0^{i-k-1} Y^k z_2 w_1$ falls in w_1 weniger als $i-k$ Einsen stehen, wird nicht akzeptiert. Somit $\ldots \overset{*}{\vdash} X^i z_1 Y^k 0 \alpha \overset{*}{\vdash} X^i Y^k z_4 0 \alpha$

gälte β): $z_1 w \overset{*}{\vdash} X^i Y^i z_4 0 w_2$ mit $w_1 \neq \varepsilon \Longrightarrow$ nicht akzeptiert

gälte γ): $z_1 w \overset{*}{\vdash} X^i Y^i z_4 1^{k-1} w_1 \Longrightarrow$ nicht akzeptiert

Definition. Eine Sprache L, zu der es eine TM T gibt, die genau L akzeptiert, d.h. für die $L(T) = L$ gilt, heißt *semientscheidbar*.
Eine Sprache L, zu der es eine TM T gibt, die immer hält (d.h. bei jeder Eingabe aus Σ^* schließlich hält), heißt *entscheidbar*.

2.1.2 TURING-Maschinen als Computer zur Berechnung von Funktionen

Definition. Die TM T *berechnet* eine n-stellige (partielle) Funktion $g : (\Sigma^*)^n \longrightarrow \{\Gamma \setminus \{\lambda\}\}^*$ g.d.w. für $w_1, \ldots, w_n \in \Sigma^*$ gilt $z_A w_1 \lambda w_2 \lambda \ldots \lambda w_n \overset{*}{\vdash} \alpha z \beta$ mit einer Endkonfiguration $\alpha z \beta$ falls $g(w_1, \ldots, w_n) = \alpha\beta$ und T hält nicht, wenn $g(w_1, \ldots, w_n)$ nicht definiert ist.

Definition. Eine Funktion $g : (\Sigma^*)^n \longrightarrow \{\Gamma \setminus \{\lambda\}\}^*$ heißt *partiell* TURING-*berechenbar*, wenn es eine TM T gibt, die g berechnet. g heißt (*total*) TURING-*berechenbar*, wenn g partiell TURING-berechenbar ist und T hält für alle Anfangskonfigurationen $z_A w_1 \lambda \ldots \lambda w_n, w_1, \ldots, w_n \in \Sigma^*$.

Beispiel. $\Sigma = \{0,1\}, n = 1$

$$g : (\Sigma^*)^n \longrightarrow \{\Gamma \setminus \{\lambda\}\}^* : g(w) = \begin{cases} 0 & \text{, falls } w \text{ gerade Anzahl Einsen enthält} \\ 1 & \text{, sonst} \end{cases}$$

besonders wichtig: Funktionen $g : \mathbb{N}^n \longrightarrow \mathbb{N}$ (auch allgemein gültig)

besonders beliebt (üblich): unäre Darstellung natürlicher Zahlen, d.h. für die natürliche Zahl n schreibt man $\underbrace{11\ldots1}_{n \text{ Einsen}}$ auf das Band

Beispiel. $+ : \mathbb{N}^2 \longrightarrow \mathbb{N}$

	1	λ
z_1	$(z_2, 1, R)$	$(z_3, 1, R)$
z_2	$(z_2, 1, R)$	$(z_3, 1, R)$
z_3	$(z_3, 1, R)$	(z_4, λ, L)
z_4	(z_5, λ, L)	$-$
z_5	$-$	$-$

2.1.3 Programmierung von Turing-Maschinen

Beispiele zeigen, dass der Entwurf einer TM für bescheidenen Aufgaben einfach ist, aber aufgrund der beschränkten Elementarfunktionen für größere Aufgaben umständlich ist.

$\rightsquigarrow$ Notwendigkeit einer „*Programmierungstechnologie*“

(1) Konstruktion von Elementarmaschinen (für „*Grundoperationen*“)

(2) Zusammensetzen von einfachen TM zu komplizierten TM

Elementarmaschinen über $\Gamma = \{\lambda, a\}$ (Beispiele)

Beispiel.

T_λ **: λ-Maschine**
$\longrightarrow$ Schreiben von λ ins Arbeitsfeld

	λ	1
z_A	(z, λ, R)	(z, λ, R)
z	(z_E, λ, L)	$(z_E, 1, L)$
z_E	-	-

T_1 **: 1-Maschine (analog)**
$\longrightarrow$ Schreiben von 1 ins Arbeitsfeld

T_r **: (kleine) Rechtsmaschine**
$\longrightarrow$ Bewegung der LSV um ein Feld nach rechts

	λ	1
z_A	(z_E, λ, R)	$(z_E, 1, R)$
z_E	-	-

T_l **: (kleine) Linksmaschine (analog)**
$\longrightarrow$ Bewegung der LSV um ein Feld nach links

T_T **: Testmaschine**
$\longrightarrow$ Hält mit Endzustand z_{E_λ}, wenn Inhalt des Arbeitsfeldes λ; Hält mit Endzustand z_{E_1}, wenn der Inhalt des Arbeitsfeldes 1

	λ	1
z_A	(z_1, λ, R)	$(z_2, 1, R)$
z_1	$(z_{E_\lambda}, \lambda, L)$	$(z_{E_\lambda}, 1, L)$
z_2	(z_{E_1}, λ, L)	$(z_{E_1}, 1, L)$
z_{E_λ}	-	-
z_{E_1}	-	-

Komposition von TM

Definition. Sind $T_1 = (\Sigma, Z_1, \Gamma, z_{A_1}, f_1, \{z_{E_1}\})$ und $T_2 = (\Sigma, Z_2, \Gamma, z_{A_2}, f_2, Z_{E_2})$ mit $Z_1 \cap Z_2 = \varnothing$ zwei TM, so versteht man unter der *Sequenz*

$$T = T_1 T_2$$

der TM T_1 und T_2 ein TM $T = (\Sigma, Z, \Gamma, z_{A_1}, f, Z_{E_2})$ mit

$$\begin{aligned} Z &:= (Z_1 \setminus \{z_{E_1}\}) \cup Z_2 \\ f &: Z \times \Gamma \longrightarrow Z \times \Gamma \times \{L, R\} \end{aligned}$$

und f entsteht durch Zusammenführen der TURING-Tafeln für f_1 und f_2, wobei im f_1-Teil die Zeile von z_{E_1} zu streichen ist und alle z_{E_1} im Werteteil durch z_{A_2} zu ersetzen sind.

(Hintereinanderausführung von T_1 und T_2 bedeutet also zunächst Anwendung von T_1 auf das im Band gespeicherte Wort und bei Erreichen des Endzustandes von T_1 Fortsetzung mit dem Anfangszustand von T_2 bis zum eventuellen Erreichen des Endzustandes von T_2)

Beispiel. Einer Gruppe von Einsen ist rechts eine 1 hinzuzufügen

$$\text{Anfangskonfiguration:} \quad 1^{m-1} z_A 1 \;:\; \ldots \lambda 111 \underbrace{1}_{z_A} \lambda \ldots$$

$$\text{Endkonfiguration:} \quad 1^{m} z_E 1 \;:\; \ldots \lambda 1111 \underbrace{1}_{z_E} \lambda \ldots$$

Offenbar ist $T = T_r T_1$.

	λ	1	
z_{A_1}	(z_{E_1}, λ, R) z_{A_2}	$(z_{E_1}, 1, R)$ z_{A_2}	T_r
z_{E_1}	-	-	
z_{A_2}	$(z, 1, R)$	$(z, 1, R)$	T_1
z	(z_{E_2}, λ, L)	$(z_{E_2}, 1, L)$	
z_{E_2}	-	-	

$\Longrightarrow$

	λ	1
z_A	(z_1, λ, R)	$(z_1, 1, R)$
z_1	$(z_2, 1, R)$	$(z_2, 1, R)$
z_2	(z_E, λ, L)	$(z_E, 1, L)$
z_E	-	-

Bemerkung.

1. Prinzip der Hintereinanderausführung von TM lässt sich sofort auf TM T_1 mit mehreren Endzuständen erweitern - dann muss ein Endzustand ausgezeichnet werden.

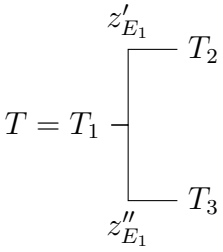
2. Spezialfall: $T = T_1T_1$; Abbkürzung für i-maliges Hintereinanderanwenden einer TM:

$$T = \underbrace{T_1 \dots T_1}_{i\text{-mal}} = T_1^i$$

Definition. Sind

$$\begin{aligned} T_1 &= (\Sigma, Z_1, \Gamma, z_{A_1}, f_1, \{z'_{E_1}, z''_{E_1}, \dots\}) \\ T_2 &= (\Sigma, Z_2, \Gamma, z_{A_2}, f_2, Z_{E_2}) \\ T_3 &= (\Sigma, Z_3, \Gamma, z_{A_3}, f_3, Z_{E_3}) \end{aligned}$$

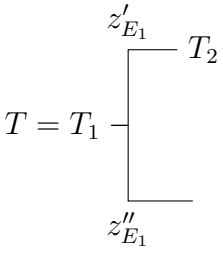
TM, so versteht man unter der *Alternative* (Verzweigung)



der TM T_2 und T_3 bezüglich T_1 eine TM

$$T = \begin{cases} T_1T_2 & \text{wenn } T_1 \text{ in } z'_{E_1} \text{ endet} \\ T_1T_3 & \text{wenn } T_1 \text{ in } z''_{E_1} \text{ endet} \\ T_1 & \text{sonst} \end{cases}$$

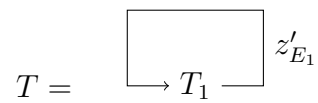
Bemerkung. Spezialfall



bedeuted

$$T = \begin{cases} T_1T_2 & \text{wenn } T_1 \text{ in } z'_{E_1} \text{ endet} \\ T_1 & \text{sonst} \end{cases}$$

Definition. Sei $T_1 = (\Sigma, Z, \Gamma, z_{A_1}, f_1, \{z_{E_1}, \dots\})$ eine TM, so versteht man unter dem Zyklus



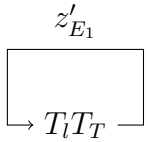
der TM T_1 die TM, die durch wiederholte Anwendung von T_1 entsteht, solange T_1 in z_{E_1} endet.

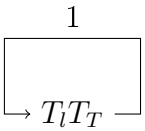
Konstruktion spezieller TM (Beispiele) $(\Gamma = \{\lambda, 1\})$

1. Große Linksmaschine T_L (Rechtsmaschine T_R)

   Aufgabe: Suchen des ersten λ's von links nach einer Gruppe von Einsen.

   Anfangskonfiguration (AK): $1^m\ z_A\ 1^l \quad , \quad l \geq 0$
   Endkonfiguration (EK): $z_E\ \lambda\ 1^{m+l}$

   $T_L =$ 

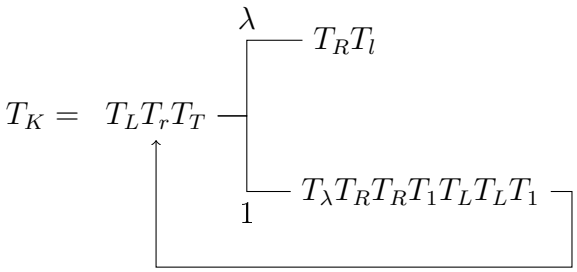
   kürzer $T_L =$ 

2. Kopiermaschine T_K für eine Gruppe von Einsen

   Aufgabe: Erzeugung einer Kopie einer Einser-Gruppe rechts neben dieser Gruppe mit Trennung durch λ.

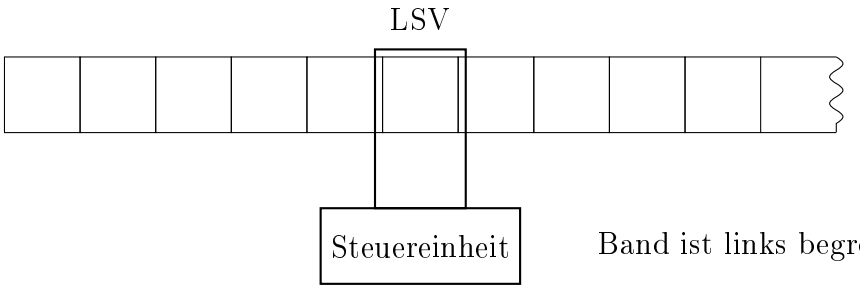
   (AK): $1^m\ z_A\ 1^l \quad , \quad l \geq 0$
   (EK): $1^{m+l}\ \lambda\ 1^{m+l-1}\ z_E\ 1$ (LSV über letzter 1)

   Prinzip: Zu kopierende 1 wird im Original vorübergehend gelöscht (durch λ ersetzt) und dann wieder reproduziert.



2.1.4 Modifikationen von TURING-Maschinen

Einseitige deterministische TM:



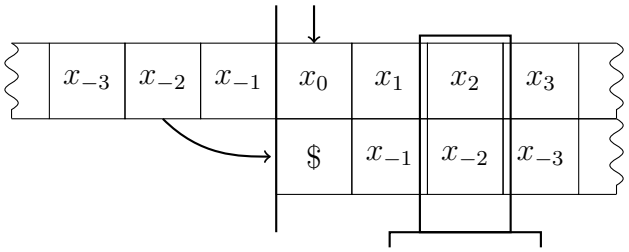
Formal wird die einseitige TM wie eine zweiseitige TM definiert, mit der Zusatzbedingung, dass die LSV nicht nach links gehen kann, wenn sie auf dem 1. Feld steht.

Satz 2.1. L wird akzeptiert von einer einseitigen TM $\Longleftrightarrow$ L wird akzeptiert von einer zweiseitigen TM.

Beweis.

$\Longrightarrow$: Konstruktion einer zweiseitigen TM, die eine vorgegebene einseitige TM simulieren kann: Sie markiert die Zelle links von ihrer Startposition und simuliert dann die einseitige TM. Wenn die markierte Zelle erreicht wird, so hält sie, ohne zu akzeptieren.

$\Longleftarrow$: Sei $T_2 = (\Sigma_2, Z_2, \Gamma_2, z_{A_2}, f_2, Z_{E_2})$ eine zweiseitige TM.



Prinzip: Die konstruierte TM (einseitige) hat 2 Spuren. Die obere Spur repräsentiert das rechte Halbband von T_2, die untere Spur repräsentiert, in umgekehrter Richtung das linke Halbband - gerechnet von der Ausgangsstellung der LSV.

$$T_2 = (\Sigma_2, Z_2, \Gamma_2, z_{A_2}, f_2, Z_{E_2})$$

Konstruktion: $T_1 = (\Sigma_1, Z_1, \Gamma_1, z_{A_1}, f_1, Z_{E_1})$

$\Sigma_1 := \Sigma_2 \times \{\lambda\}$; auf obere Spur Zeichen von Σ_2, auf untere Spur λ

$Z_1 := (Z_2 \times \{O, U\}) \cup \{z_{A_1}\}$; O für obere Spur, U für untere Spur

$\Gamma_1 := \Gamma_2 \times (\Gamma_2 \cup \{\$\})$ $\$ \notin \Gamma_2$, \$ markiert (auf der unteren Spur) die 1. Zelle

$Z_{E_1} := \{(z, O), (z, U) : z \in Z_{E_2}\}$

f_1:

1. Im 1. Schritt gehe T_2 nach rechts:
   Dann
   - \$ schreiben
   - O setzen (obere Spur)
   - nach rechts
   - schreibt oben wie T_2

   d.h $\forall a \in \Sigma_2 \cup \{\lambda\}$

   $$f_1(z_{A_1}, (a, \lambda)) = ((z, O), (X, \$), R) \text{ , wenn } f_2(z_{A_2}, a) = (z, X, R)$$

2. Im 1. Schritt gehe T_2 nach links:

   $$f_1(z_{A_1}, (a, \lambda)) = ((z, U), (X, \$), R) \text{ , wenn } f_2(z_{A_2}, a) = (z, X, L)$$

   (analog zu oben - untere Spur)

3. „*Simulation auf oberer Spur*“

   $$f_1((z, O), (X, Y)) = ((z', O), (X', Y), A) \text{ , wenn } f_2(z, X) = (z', X', A)$$

   $A = L$ bzw. $A = R, Y \neq \$$

4. „*Simulation auf unterer Spur*“

$$f_1((z,U),(X,Y)) = ((z',U),(X,Y'),\bar{A}) \text{ , wenn } f_2(z,Y) = (z',Y',A)$$

$A = L$ bzw. $A = R, Y \neq \$$
$\bar{A} = L$ bzw. $\bar{A} = R$

5. „*Simulation an Startposition*“

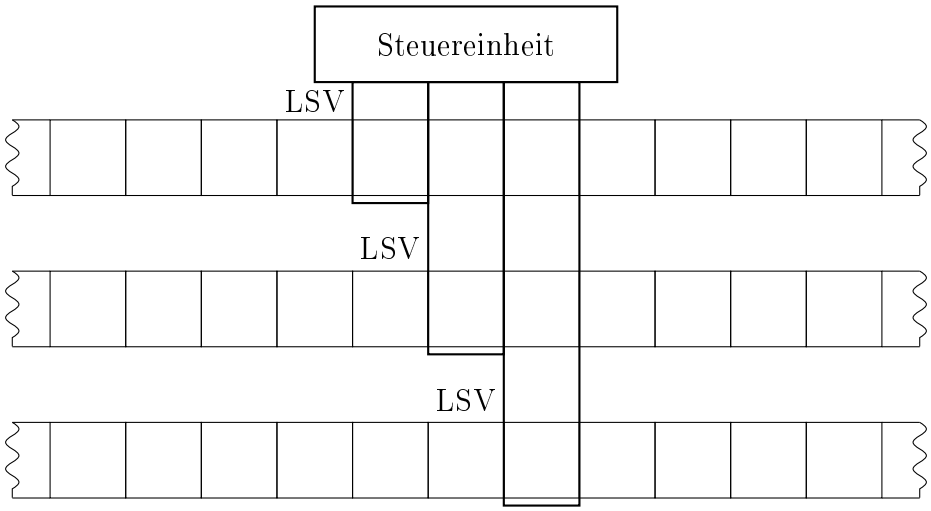
$$f_1((z,O),(X,\$)) = ((z,U),(X,\$)) = ((z',C),(Y,\$),R) \text{ , wenn } f_2(z,X) = (z',Y,A)$$

$A = L$ bzw. $A = R$

$$C = \begin{cases} O & \text{, wenn } A = R \\ U & \text{, wenn } A = L \end{cases}$$

Offensichtlich akzeptieren T_1 und T_2 die gleiche Sprache! □

TM mit mehreren Bändern
Das Modell



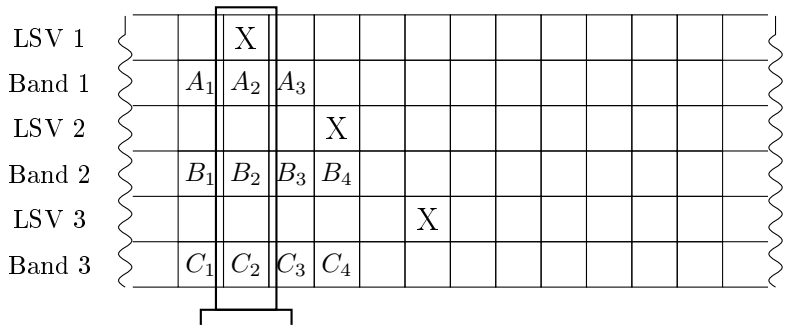
besteht aus Steuereinheit (mit endlich vielen Zuständen) und $k \geq 1$ Bänder (zweiseitig unendlich) sowie k LSVs.
In einem Schritt - abhängig vom Zustand und allen gelesenen Symbolen - kann die TM

- einen (neuen) Zustand annehmen
- ein Zeichen in die Zellen schreiben, über denen eine LSV steht
- die LSVs (unabhängig von einander) nach rechts oder links bewegen oder festhalten
- am Anfang steht das Input auf einem Band und die anderen sind leer. (LSV über 1. Zeichen)

$$f : Z \times \Gamma^k \longrightarrow Z \times \Gamma^k \times \{L,R,O\}^k \qquad (O \text{ - „}\textit{festgehalten}\text{“})$$

Satz 2.2. Wird eine Sprache L akzeptiert von einer TM T_1 mit k Bändern, so wird sie akzeptiert von einer TM mit einem Band.

Beweis. Werde L akzeptiert von einer TM T_1 mit k Bändern.
Konstruktion einer TM T_2 mit einem Band und $2k$ Spuren, 2 Spuren für jedes Band von T_1:
Eine Spur enthält den Inhalt des entsprechenden Bandes von T_1, die andere ist leer, außer einer Marke, die die Stellung der LSV für dieses Band markiert.



Die Steuereinheit von T_2 speichert den Zustand von T_1, die *Anzahl* der LSV-Markierungen rechts von der LSV von T_2 und ein k-Tupel von Bandsymbolen und ein k-Tupel von Kopfbewegungen. Jeder Schritt von T_1 wird simuliert durch ein Überstreichen des Bandes von T_2 von links nach rechts und zurück:
Anfangs ist die LSV von T_2 über der Zelle am Weitesten links, die eine Markierung enthält. Dann geht die LSV nach rechts, wobei beim Antreffen einer Zelle mit Markierung das dazu gehörige Zeichen gemerkt wird und die Anzahl der LSV-Markierungen rechts von T_2's LSV entsprechend abgeändert wird.
Wenn sich keine markierte Zelle rechts von T_2's LSV befindet, so „*kennt*“ T_2 alle von T_1 gelesenen Bandsymbole und bestimmt den Schritt von T_1, d.h. das k-Tupel von zu schreibenden Zeichen und das k-Tupel der Bewegung der LSV's von T_1. Dann geht die LSV von T_2 nach links bis die am Weitesten links stehende Markierung erreicht ist, wobei sie beim Passieren einer Markierung das entsprechende Zeichen in die entsprechende Zelle schreibt und die entsprechende Markierung nach links oder rechts verschiebt bzw. fest lässt - je nachdem, welche Bewegung die entsprechende LSV von T_1 ausführt. □

Nichtdeterministische TM

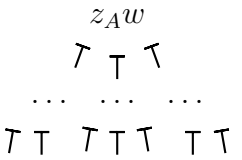
Definition. Eine (zweiseitige) nichtdeterministische TM

$$T = (\Sigma, Z, \Gamma, z_A, R_f, Z_E)$$

ist bestimmt durch $\Sigma, Z, \Gamma, z_A, Z_E$ wie oben und

$$R_f \subseteq Z \times \Gamma \times Z \times \Gamma \times \{R, L\}$$

Alle Begriffe wie Konfiguration, $\vdash$, Endkonfiguration, akzeptierte Sprache sind analog zu übertragen.



Konfigurationenbaum

Wie bei endlichen Automaten - im Gegensatz zu KA - ergibt sich hier, dass „durch den Nichtdeterminismus nichts gewonnen wird.“

Satz 2.3. Wird L akzeptiert von einer NTM T_1, so wird L akzeptiert von einer TM T_2.

Beweisprinzip: T_2 probiert alle Konfigurationen von T_1 durch (*Breite-Zuerst-Suche* im Konfigurationenbaum).

Bemerkung. Es gibt zahlreiche weitere Modelle deterministischer und nichtdeterministischer TM, die alle „*dasselbe*“ leisten.

2.1.5 TURING-Maschinen und Sprachen

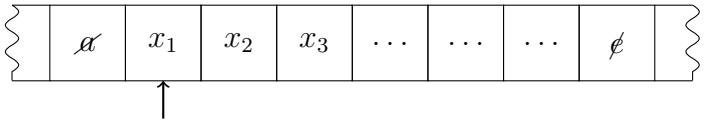
Bezüglich der von uns betrachteten Sprachfamilien $\mathcal{L}_i (i = 0, 1, 2, 3)$ hatten wir:

$\mathcal{L}_3 \equiv$ die von endlichen Automaten akzeptierten Sprachen
$\mathcal{L}_2 \equiv$ die von endlichen nichtdeterministischen KA akzeptierten Sprachen

Welche Automaten gehören zu $\mathcal{L}_1, \mathcal{L}_0$?

Definition. Ein linear beschränkter Automat (LBA) ist eine *nichtdeterministische* TM mit folgenden Eigenschaften:

- Das Eingabealphabet besitzt zwei spezielle Symbole $\not\alpha$ und $\not\epsilon$, die Anfang bzw. Ende markieren.
- Beim Lesen von $\not\alpha$ wird wieder $\not\alpha$ geschrieben und der LBA kann nicht nach links gehen; beim Lesen von $\not\epsilon$ wird wieder $\not\epsilon$ gechrieben und der LBA kann nicht nach rechts gehen.



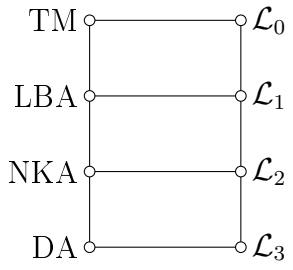
kann sich zwischen $\not\alpha$ und $\not\epsilon$ nicht ausdehnen, d.h. LBA

Die von einem LBA T akzeptierte Sprache ist

$$L(T) = \{w : w \in (\Sigma \setminus \{\not\alpha, \not\epsilon\})^* : z_A \not\alpha w \not\epsilon \overset{*}{\vdash} \alpha z \beta, z \in Z_E\}$$

Satz 2.4. Ist $L \in \mathcal{L}_1$, so gibt es einen LBA M mit $L = L(M)$.
Für jeden LBA M ist $L(M) \in \mathcal{L}_1$. (o.B.)

Satz 2.5. Ist $L \in \mathcal{L}_0$, so gibt es eine TM T mit $L = L(T)$.
Für jede TM T gilt $L(T) \in \mathcal{L}_0$. (o.B.)



2.1.6 Universelle TURING-Maschinen

Bislang: Verschiedene Algorithmen werden durch verschiedene TM realisiert.

Frage. Lässt sich eine TM ausgeben, die in einem gewissen Sinn jeden Alogrithmus verwirklichen und damit die Arbeit jeder speziellen TM „*simmulieren*“ kann?

Simulation einer TM durch einen Menschen:
Gegeben: TURING-Tafel + Startkonfiguration

A1) Suche in der Konfiguration das Zustandszeichen und „*merke*“ das dahinter stehende Bandsymbol.

A2) Suche in der Maschinentafel (TURING-Tafel) die Zeile mit dem gefundenen Zustandszeichen und die Spalte mit dem gemerkten Bandzeichen.

A3) Gibt es keinen Eintrag an dieser Stelle, so ist der Prozess beendet.

A4) Andernfalls notiere das dort stehende Tripel.

A5) Ist das 3. Zeichen des notierten Tripels ein L, so ersetzte man in der Konfiguraton das Zeichen rechts vom Zustandszeichen durch das 2. Zeichen des notierten Tripels; das Zustandszeichen durch das Zeichen links vom Zustandszeichen und das Zeichen links vom Zustandszeichen durch das 1. Zeichen des notierten Tripels.

A6) Ist das 3. Zeichen des notierten Tripels ein R, so ... (analog).

A7) Setzte A1) fort.

Eine TM, die diesen Prozess ebenso nachvollziehen kann, also eine beliebige TM simulieren kann, wird als *universelle* TM bezeichnet.

Problem bei der Konstruktion:

- Die von einer universellen TM T_u zu verarbeitende Information besteht aus:

  (1) Maschienentafel der zu simulierenden TM

  (2) Startkonfiguration, die von der zu simulierenden TM zu verarbeiten ist

  $\Longrightarrow$ zweidimensionale Maschienentafel muss eindimensional gestaltet werden

- T_u hat wie jede TM eine feste Anzahl von Zuständen und Bandzeichen und muss damit TM mit beliebig großen Bandalphabeten und Zustandsmengen simulieren können.

$$\begin{aligned} T &= (\Sigma, Z, \Gamma, z_A, f, Z_E) \\ T_u &= (\Sigma_u, Z_u, \Gamma_u, z_{A_u}, f_u, Z_{E_u}) \end{aligned}$$

Darstellung der Maschienentafel von T als Element von Σ_u^*:

(1) Anordnung der Informationen als Folge von Quintupeln $(Z_\nu, X_\mu, Z_i, X_i, Z_k)$ mit Reihenfolge ohne Trennzeichen, z.B.:

	0	1	λ
z_1	(λ, z_1, R)	(λ, z_2, R)	$(0, z_3, R)$
z_2	(λ, z_2, R)	(λ, z_1, R)	$(1, z_3, R)$
z_3	-	-	-

$$z_1 0 \lambda z_1 R z_1 1 \lambda z_2 R z_1 \lambda 0 z_3 R z_2 0 \lambda z_2 R z_2 1 \lambda z_1 R z_2 \lambda 1 z_3 R z_3 0 z_3 1 z_3 \lambda$$

(2) Annahme $\Sigma_u = \{0, 1\}; \Gamma_u = \{\lambda, 0, 1, \ldots\}$ mit *endlich* vielen zusätzlichen Zeichen. Codieren der Quintupel durch Σ_u:

$$\begin{aligned} X_i \in \Gamma (i = 1, \ldots, n) &: \underbrace{100\ldots01}_{2i+2} = 10^{2i+2}1 \\ Z_j \in Z (j = 1, \ldots, k) &: 10^{2j+1}1 \\ L &: 101 \\ R &: 1001 \end{aligned}$$

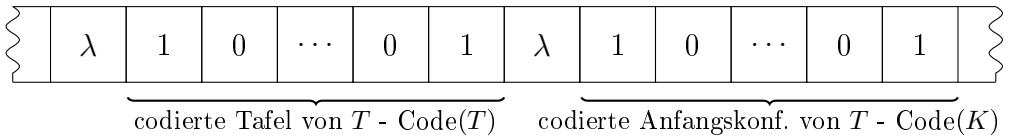
Beispiel. $\Gamma = \{\lambda, 0, 1\} = \{X_1, X_2, X_3\}$, $Z = \{z_1, z_2, z_3\}$

$$z_1 0 \lambda z_1 R \longmapsto 10^3 110^6 110^4 110^3 110^2 1$$

Darstellung der Konfiguration von T als Element von Σ_u^*:
Codierung wie oben, wobei die Leerzeichen links und recht der Konfiguration nicht codiert werden.

Entwurf von T_u:

- Anfangskonfiguration:



  Konstruktion von f_u nicht im Detail. Formulieren des Simulationsalgorithmus in angepasster Form.

Funktionsweise von T_u:

A1) Suche in Code(K) die (eindeutig bestimmte) Codegruppe mit ungerader Anzahl Nullen.

A2) Suche in Code(T) das Paar benachbarter Codegruppen, welche mit dem Codegruppenpaar in Code(K) übereinstimmt, in dem die in A1) ermittelte Codegruppe 1. Element ist!

⋮

Die so modifizierten Anweisungen lassen ich ersichtlich in das Programm einer TM umsetzen. Am Ende der Arbeit von T_u (wenn es eines gibt) steht auf dem Bandbereich von Code(K) die (codierte) Endkonfiguration von T.
⇝

Satz 2.6. Es gibt eine universelle TM (die jede beliebige TM simulieren kann).
(es gibt eine solche mit fünf Zuständen und sieben Bandsymbolen)

Bemerkung. Eine universelle TM kann als Modell eines programmgesteuerten Rechenautomaten angesehen werden. (Speicher enthält die Inputdaten für die gestellten Aufgaben und das zu ihrer Lösung benötigte Programm.)

2.2 Entscheidbarkeit

2.2.1 Probleme und Sprachen

Wiederholung: Eine Sprache L heißt entscheidbar, wenn es eine TM gibt, die für jedes Input hält und die genau L akzeptiert.

Definition (*Problem*)**.** Ein Problem ist eine Fragestellung mit Ja/Nein-Antwort, die sich auf (endlich viele) Parameter bezieht.

z.B. Allgemeines Halteproblem für TM: Hält (eine beliebige TM) T mit (beliebigem) Input w (über dem Eingabealphabet von T)?
Dieses Problem hat 2 Parameter, T und w.

Definition (*entscheidbares Problem*)**.** Ein Problem heißt entscheidbar, wenn es einen terminierenden Algorithmus gibt, der für jede mögliche spezielle Argumentation (aktuelle Parameter) entscheidet, ob die Antwort „*Ja*“ oder „*Nein*“ ist (bzw. 1 oder 0).

z.B. Entscheidbarkeit des allg. Halteproblems: Gibt es einen *terminierenden Algorithmus*, der für **jede** TM T und für **jedes** Input $w \in \Sigma^*$ als Ergebnis 1 liefert, wenn T mit Input w hält und sonst 0?

Kann diese Frage mit „*Ja*“ beantwortet werden, so ist das allgemeine Halteproblem entscheidbar; kann diese Frage mit „*Nein*“ beantwortet werden, so ist das allgemeine Halteproblem nicht entscheidbar (unentscheidbar).

Der Begriff „*terminierender Algorithmus*“ ist jetzt zu ersetzen durch eine TM, die für jedes Input über ihrem Eingabealphabet hält.
Dann hat man das Input für die TM - also die Argumente - in geeigneter Weise zu codieren.

Die codierten Argumente, für die die Antwort „*Ja*“ ist (ein k-Tupel, wenn das Problem k Parameter hat), bilden eine Sprache (über dem Eingabealphabet der TM) - die Sprache $L(P)$ des Problems P.

offensichtlich gilt:
Das Problem P ist entscheidbar, wenn seine Sprache $L(P)$ entscheidbar ist.

Beispiel. Für das allgemeine Halteprobem kann man als Codierung von $(T, z_A w)$ z.B. die Codierung des vorherigen Abschnittes nehmen und erhält als Sprache die Codes, bei denen T auf w angewandt hält.

Beispiel. Primzahlenproblem P: Ist $n \in \mathbb{N}$ prim?

$$L(P) = \{1^n : \ n \text{ prim}\}$$

2.2.2 Existenz nicht entscheidbarer Probleme

Zunächst behandeln wir Probleme über TM. Da wir die Nichtentscheidbarkeit zeigen werden, können wir uns auf TM mit $\Sigma = \{0, 1\}$ und $\Gamma = \{0, 1, \lambda, \ldots\}$ beschränken, denn ist ein spezielles Problem nicht entscheidbar, so erst recht ein Allgemeineres.
Ferner seien TM wie oben codiert und für (T, w) mit TM T und $w \in \{0, 1\}^*$ sei die Codierung

$$\text{Code}(T)\ 11w$$

Definition. Eine TM T (mit $\{0, 1\} \subseteq \Sigma$) heißt selbstanwendbar, wenn T auf Code(T) angewandt hält.

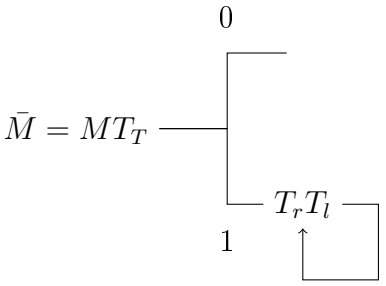
Selbstanwendbarkeitsproblem: Ist T selbstanwendbar?

Nochmals genauer formuliert:

$(*)$ Gibt es eine TM M, die für jedes Input der Form Code(T) hält und zwar mit Output 1, wenn T selbstanwendbar ist und mit Output 0, wenn T nicht selbstanwendbar ist?

Satz 2.7. Das Selbstanwendbarkeitsproblem ist nicht entscheidbar.

Beweis. Wäre das Problem entscheidbar, so gäbe es M mit $(*)$ und o.B.d.A. mit Endkonfiguration z_E 1 bzw. z_E 0



d.h.

- $\bar{M}$ hält nicht, wenn angewandt auf Code(T) mit T selbstanwendbar.
- $\bar{M}$ hält mit Output 0, wenn angewandt auf Code(T) mit T nicht selbstanwendbar.

Wir betrachten $\bar{M}$.

- Wäre $\bar{M}$ selbstanwendbar $\Longrightarrow$ (nach Definition von $\bar{M}$), $\bar{M}$ hält nicht, wenn angewandt auf Code($\bar{M}$), d.h. $\bar{M}$ ist nicht selbstanwendbar. ↯
- Wäre $\bar{M}$ nicht selbstanwendbar, so hält $\bar{M}$ angewandt auf Code($\bar{M}$), d.h. $\bar{M}$ ist selbstanwendbar. ↯

Also muss die Annahme, das Problem wäre entscheidbar, **falsch** sein! □

Satz 2.8. Das allgemeine Halteproblem ist nicht entscheidbar.

Beweis. Angenommen, es gäbe eine TM M, die das allgemeine Halteproblem entscheidet, d.h. für beliebige TM T und Wort w über dem Eingabealphabet von T leistet M folgendes:

$$\text{Code}(T)11w \longmapsto \begin{cases} 1 & \text{, wenn } T \text{ auf } w \text{ angewandt hält} \\ 0 & \text{, wenn } T \text{ auf } w \text{ angewandt nicht hält} \end{cases}$$

Es gibt eine TM H, die für jede TM folgendes leistet:

$$\text{Code}(T) \longmapsto \text{Code}(T)11\text{Code}(T)$$

Die TM HM überführt dann

$$\text{Code}(T) \longmapsto \begin{cases} 1 & \text{, wenn } T \text{ auf Code}(T) \text{ hält} \\ 0 & \text{, wenn } T \text{ auf Code}(T) \text{ nicht hält} \end{cases}$$

d.h. HM entscheidet das Selbstanwendbarkeitsproblem. ↯ □

Bemerkung. Nimmt man als Explikat des Algorithmusbegriffes den eines C-Programmes - was man tun kann - so kann man formulieren:

> „Es gibt kein C-Programm, mit dem man testen könne, ob ein *beliebiges* C-Programm auf beliebige zulässige Daten angewandt, terminiert."

2.2.3 Weitere nicht entscheidbare Probleme

Postsches Korrespondenzproblem (PKP)
Sei Σ ein Alphabet und

$$P = (p_1, \ldots, p_k) \ , \ Q = (q_1, \ldots, q_k)$$

Listen von Strings über Σ.
Gibt es Indizes $i_1, \ldots, i_m$ mit $m \geq 1$, so dass

$$p_{i_1} p_{i_2} \ldots p_{i_m} = q_{i_1} q_{i_2} \ldots q_{i_m} \ ?$$

Wenn ja, so heißt $(i_1, \ldots, i_m)$ eine Lösung des PKP.
Es ist die Frage, ob das PKP (mit bel. Σ, P, Q) entscheidbar ist.

Beispiel. $\Sigma = \{0, 1\}$

$$P = (1, 10111, 10) \ , \ Q = (111, 10, 0)$$

$$p_2 p_1 p_1 p_3 = \underline{10}\ \underline{111}\ \underline{111}\ \underline{0} = q_2 q_1 q_1 q_3$$

D.h. (2,1,1,3) ist eine Lösung (dieses speziellen) PKP.

Bemerkung. Ein Versuch, der Sache durch systematisches Probieren beizukommen, scheitert: Wenn man eine Lösung findet, so ok; wenn nicht, so weiß man nicht ob man abbrechen kann (Semientscheidbarkeit).

Der Nachweis der Unentscheidbarkeit des PKP gelingt durch Zurückführung auf die Unentscheidbarkeit des allgemeinen Halteproblems.

Probleme für kontextfreie Grammatiken
Seien G_1 und G_2 beliebige kontextfreie Grammatiken, L_r eine beliebige reguläre Sprache (gegeben etwa durch eine reguläre Grammatik) (immer gleiches Terminalalphabet).
Folgende Probleme sind nicht entscheidbar:

(a) $L(G_1) \cap L(G_2) = \varnothing$?

(b) Ist G_1 eindeutig?

(c) Ist $L(G_1) = T^*$?

(d) Ist $L(G_1) = L(G_2)$?

(e) Ist $L(G_1) \subseteq L(G_2)$?

(f) Ist $L(G_1) = L_r$?

(g) Ist $L_r \subseteq L(G_1)$?

Beweis gelingt durch Zurückführung auf Nichtentscheidbarkeit des PKP.

Bemerkung (1). Wenn G_1 und G_2 regulär, so sind alle diese Probleme entscheidbar.

Bemerkung (2). Wenn G_1 kontextfrei, $L(G_1) = \varnothing$ entscheidbar.

2.3 Berechenbarkeit

2.3.1 Rekursive Funktionen

Wir betrachten arithmetische Funktionen, d.h. Funktionen

$$f : \mathbb{N}^n \longrightarrow \mathbb{N}$$

Nach Abschnitt 2.1 ist klar, was eine TURING-berechenbare Funktion ist.
Diese Definition ist sehr flexibel für eine mathematische Handhabung. Wir wollen einen neuen Begriff für Berechenbarkeit prägen - ein neues Explikat des Algorithmusbegriffes.

Frage. Gibt es überhaupt Funktionen (immer $\mathbb{N}^n \to \mathbb{N}$), die nicht TURING-berechenbar sind?

Satz 2.9. Es gibt nur abzählbar viele TM und damit nur abzählbar viele einstellige TURING-berechenbare Funktionen.

Beweis. Es gibt nur endlich viele TM mit $|Z| = k$ und $|\Gamma| = l$. Diese Menge sei $M(k, l)$. Dann erhält man in der Folge

$$M(1,1), M(1,2), M(2,1), M(3,1), M(2,2), M(1,3), \ldots$$

alle TM, man kann sie also durchnummerieren. □

Satz 2.10. Es gibt überabzählbar viele einstellige arithmetische Funktionen.

Beweis. Angenommen, es gäbe *nur abzählbar viele*; so sei $f_1, f_2, f_3, \ldots$ die Menge aller (einstelligen) arithmetischen Funktionen.
Sei $f : \mathbb{N} \longrightarrow \mathbb{N}$ wie folgt definiert:

$$f(k) := \begin{cases} 0 & \text{, wenn } f_k(k) \neq 0 \\ 1 & \text{, wenn } f_k(k) = 0 \end{cases}$$

f kommt unter $f_1, f_2, \ldots$ nicht vor, denn $f(k) \neq f_k(k)$ ↯
(CANTORsches Diagonalverfahren) □

2.3.2 Primitiv-rekursive Funktionen

Grundlegende mathematische Methoden zur Definition neuer Funktionen:

- Substitution
- induktive Definition (Rekursion)

Definition. Sind $h_i (i = 1, \ldots, r)$ n-stellige Funktionen und ist g eine r-stellige Funktion, so versteht man unter *Substitution* die Festlegung einer n-stelligen Funktion f in der Form

$$f(x_1, \ldots, x_n) = g\left(h_1(x_1, \ldots, x_n), \ldots, h_r(x_1, \ldots, x_n)\right)$$

für beliebige $x_i \in \mathbb{N}$.

Definition. Sind g eine n-stellige und h eine $(n+2)$-stellige Funktion, so heißt die $(n+1)$-stellige Funktion f induktiv definiert durch g und h, wenn für beliebige $x_1, \ldots, x_n, y \in \mathbb{N}$ gilt:

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\ f(y+1, x_1, \ldots, x_n) &= h(y, f(y, x_1, \ldots, x_n), x_1, \ldots, x_n) \end{aligned}$$

(Induktionsschema, Rekursionsschema)

Beispiel (1).

$$\begin{aligned} f(0) &= 1 \\ f(y+1) &= h(y, f(y)) \text{ mit } h(u,v) = (u+1)\cdot v \end{aligned}$$

$$\begin{aligned} \text{also} \quad f(0) &= 1 \\ f(y+1) &= (y+1)\cdot f(y) \\ \text{d.h.} \quad f(y) &= y! \text{ (hier } n=0) \end{aligned}$$

Beispiel (2).

$$\begin{aligned} f(0,x) &= g(x) \text{ mit } g(x) = x \\ f(y+1,x) &= h(y, f(y,x), x) \text{ mit } h(u,v,w) = (u+1)\cdot v \cdot w \end{aligned}$$

$$\begin{aligned} \text{also } f(0,x) &= x \\ f(y+1,x) &= (y+1)\cdot f(y,x)\cdot x \\ &= (y+1)\cdot x\cdot y\cdot x\cdot \ldots \cdot 2\cdot x\cdot 1\cdot x\cdot x \\ &= (y+1)!\; x^{y+2} \end{aligned}$$

Frage. Welche Funktionen lassen sich durch wiederholte Anwendung des Substitutions- und Induktionsvorganges aus einem geeignet gewählten Satz von Startfunktionen erhalten?

Ausgangsfunktion

(a) Nachfolgerfunktion $N(x) = x+1$

(b) Identitätsfunktion (Projektionen) $U_n^i(x_1,\ldots,x_n) = x_i$ $(n \geq 1)$

(c) Konstante Funktion $C_n^q(x_1,\ldots,x_n) = q = const.$ $(n \geq 2)$

Definition. Eine n-stellige Funktion f heißt *primitiv-rekursiv*, wenn sie

(1) eine der Ausgangsfunktionen U_n^i, C_n^q, N ist
(bei $f = N, n = 1$)

(2) aus diesen Ausgangsfunktionen durch endlich viele Anwendungen des Substitutions- und/oder des Induktionsprozesses erhalten werden kann.

Beispiel (1). $S(y,x) = y + x$:

$$\begin{aligned} f(0,x) &= g(x) \text{ mit } g(x) = x \\ f(y+1,x) &= h(y, f(y,x), x) \text{ mit } h(u,v,w) = v+1 \\ &= f(y,x)+1 \end{aligned}$$

und $g(x) = U_1^1(x),\ h(u,v,w) = N(U_3^2(u,v,w))$
also ist S primitiv-rekursiv.

Beispiel (2). $P(y,x) = y\cdot x$:

$$\begin{aligned} f(0,x) &= g(x) \text{ mit } g(x) = 0 \\ f(y+1,x) &= h(y, f(y,x), x) \text{ mit } h(u,v,w) = v+w \\ &= f(y,x)+x \end{aligned}$$

und $g(x) = C_1^0(x),\ h(u,v,w) = S(U_3^2(u,v,w), U_3^3(u,v,w))$
also ist $P(y,x)$ primitiv-rekursiv.

Beispiel (3). x^y :

$$\begin{aligned} x^0 &= 1 \\ x^{y+1} &= x^y \cdot x \end{aligned}$$

also x^y primitiv-rekursiv.

Beispiel (4). Vorgängerfunktion

$$V(x) = \begin{cases} 0 & \text{, für } x = 0 \\ x - 1 & \text{, sonst} \end{cases}$$

$$\begin{aligned} V(0) &= 0 \\ V(y+1) &= y \end{aligned}$$

Beispiel (5). arithmetische Differenz $x - y$

$$x - y = \begin{cases} 0 & \text{, für } x \leq y \\ x - y & \text{, sonst} \end{cases}$$

$$\begin{aligned} x - 0 &= x \\ x - (y+1) &= V(x - y) \end{aligned}$$

Beispiel (6).

$$\overline{sg}(x) = \begin{cases} 0 & \text{, für } x > 0 \\ 1 & \text{, für } x = 0 \end{cases}$$

$$\overline{sg}(x) = 1 - x$$

Beispiel (7).

$$sg(x) = \overline{sg}(\overline{sg}(x))$$

Bemerkung. Beispiele demonstrieren, dass in den allgemeinen Schemata für Substitution und Induktion auch spezielle Fälle enthalten sind (z.B. Anzahl der Argumente in den Funktionen für Substitution unterschiedlich, ... $\longrightarrow$ Anwendung von $U_n^i(x_1, \ldots x_n)$).

Spezielles Resultat:

Satz 2.11. Wenn die Funktion f primitiv-rekursiv ist, so sind es auch die „*endlichen Summen und Produkte*“.

$$\begin{aligned} S(y, x_1, \ldots, x_n) &= \sum_{i=0}^{y} f(i, x_1, \ldots, x_n) \\ P(y, x_1, \ldots, x_n) &= \prod_{i=0}^{y} f(i, x_1, \ldots, x_n) \end{aligned}$$

Beweis.

$$\begin{aligned} S(0, x_1, \ldots, x_n) &= f(0, x_1, \ldots, x_n) \\ S(y+1, x_1, \ldots, x_n) &= S(y, x_1, \ldots, x_n) + f(y+1, x_1, \ldots, x_n) \end{aligned}$$

(für P analog) □

Satz 2.12. Jede primitiv-rekursive Funktion ist TURING-berechenbar.

Beweis. Wegen der Definition der primitiv-rekursiven Funktionen genügt es zu zeigen, dass

(a) die Grundfunktionen TURING-berechenbar sind

(b) durch Substitution und Induktion aus TURING-berechenbaren Funktionen wieder TURING-berechenbare Funktionen entstehen.

(a): klar!

(b): **Substitution:** Sei

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_r(x_1, \ldots, x_n))$$

und h_i und g TURING-berechenbar.
Kontruktion einer TM T, die f berechnet:

- T hat r Bänder
- auf Band 1 steht $x_1, \ldots, x_n$
- T kopiert $x_1, \ldots, x_n$ auf alle Bänder
- T berechnet auf Band 1 $h_1(x_1, \ldots, x_n), \ldots,$ auf Band r $h_r(x_1, \ldots, x_n)$
- dann kopiert T die Inhalte von Band 2, ... , Band r auf Band 1 hinter dessen Inhalt und berechnet g auf Band 1

Induktion: Sei

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\ f(y+1, x_1, \ldots, x_n) &= h(y, f(y, x_1, \ldots, x_n), x_1, \ldots, x_n) \end{aligned}$$

und g und h TURING-berechenbar.
Konstruktion von TM T:

- auf Band 1 stehen $y, x_1, \ldots, x_n$
- T schreibt 0 auf Band 2 und kopiert $x_1, \ldots, x_n$ auf Band 3 und berechnet g auf Band 3
- solange Zahl auf Band 2 $< y$ macht T folgendes:
  - es kopiert Zahl von Band 2 auf Band 3 vor dessen Inhalt
  - kopiert $x_1, \ldots, x_n$ auf Band 3 nach dessen Inhalt
  - addiert 1 zu Zahl auf Band 2 und berechnet h auf Band 3
  - Ergebnis am Ende auf Band 3

□

2.3.3 Die ACKERMANN-Funktion

Beispiele zeigen, dass viele Funktionen der mathematichen Praxis primitiv-rekursiv sind.

Frage. Ist jede brechenbare Funktion primitiv-rekursiv? (HILBERT 1926)

Antwort. Nein! (ACKERMANN 1928)

Idee. Konstruktion einer berechenbaren Funktion, die schneller wächst, als jede primitiv-rekursive Funktion.

Fortsetzung der Funktionenfolge Summe, Produkt, Potenz:

$$\begin{aligned} f_0(y,x) &= N(y) \\ f_1(y,x) &= y+x \\ f_2(y,x) &= y*x \\ f_3(y,x) &= x^y \end{aligned}$$

mit

$$\begin{aligned} f_1(0,x) = x \quad &, \quad f_1(y+1,x) = f_0(f_1(y,x),x) \\ f_2(0,x) = 0 \quad &, \quad f_2(y+1,x) = f_1(f_2(y,x),x) \\ f_3(0,x) = 1 \quad &, \quad f_3(y+1,x) = f_2(f_3(y,x),x) \end{aligned}$$

$\rightsquigarrow$ allgemeines Schema

$$\begin{aligned} f_{n+1}(0,x) &= g_{n+1}(x) \\ f_{n+1}(y+1,x) &= f_n(f_{n+1}(y,x),x) \end{aligned}$$

Wenn $g_{n+1}(x)$ eine Folge von primitiv-rekursiven Funktionen ist, so ist jede Funktion f_n primitiv-rekursiv und diese extrapolieren für $n \geq 4$ die Funktionen Summe, Produkt, Potenz.

Entscheidbarer Schritt: Ersetzen durch unendliche Folge von Funktionen $f_n(y,x)$ $(n = 0,1,\ldots)$ von 2 Argumenten durch **eine** Funktion von 3 Argumenten:

$$f(n,y,x) = f_n(y,x)$$

$f(n,y,x)$ erfüllt die Funktionalgleichung

$$f(n+1,y+1,x) = f(n,f(n+1,y,x),x)$$

Diese zusammen mit Gleichungen für verschiedenes 1. und 2. Argument liefern ein allgemeineres Induktionsschema als das der primitiven Rekursion.
Nach Reduktion durch Weglassen von x, das nur Parameterrolle spielt. Dann Ersetzen von n durch x.
ACKERMANNsche Funktion:

$$\begin{aligned} f(0,y) &= y+1 \\ f(x+1,0) &= f(x,1) \\ f(x+1,y+1) &= f(x,f(x+1,y)) \end{aligned}$$

Es gibt genau eine Funktion, die diesen Gleichungen genügt und diese Funktion ist berechenbar. (Vollständige Induktion nach x) z.B.

1. $f(2,1)$
2. $f(1,f(2,0))$
3. $f(1,f(1,1))$
4. $f(1,f(0,f(1,0)))$

5. $f(1, f(0, f(0, 1)))$
6. $f(1, f(0, 2))$
7. $f(1, 3)$
8. $f(0, f(1, 2))$
9. $f(0, f(0, f(1, 1)))$
10. $f(0, f(0, f(0, f(1, 0))))$
11. $f(0, f(0, f(0, f(0, 1))))$
12. $f(0, f(0, f(0, 2)))$
13. $f(0, f(0, 3))$
14. $f(0, 4)$
15. 5

Satz 2.13. Die ACKERMANNsche Funktion ist **nicht** primitiv-rekuriv

Beweisprinzip Man beweist zuächst eine Reihe von Ungleichungen (z.B. $y < f(x, y)$) und mit deren Hilfe folgende Eigenschaften von f:
Zu jeder primitiv-rekursiven Funktion $g(x_1, \ldots, x_n)$ gibt es eine Zahl c derart, dass $\forall x_1, \ldots, x_n$ gilt:

$$(\#) \qquad g(x_1 \ldots, x_n) < f(c, x_1 + \ldots + x_n)$$

(Man zeigt $(\#)$ für die Ausgangsfunktionen und dass sich $(\#)$ bei Substitution und Induktion „*überträgt*“)
Dann geht der Beweis so:
Wäre $f(x, y)$ primitiv-rekursiv, so auch $g(x) = f(x, x)$. Nach $(\#)$ gibt es ein c mit

$$g(x) < f(c, x)\ \forall x$$

Insbesondere für $x = c$:

$$g(c) < f(c, c) = g(c) \qquad \lightning$$

Es gilt:

$$\begin{aligned} f(1, y) &= y + 2 \\ f(2, y) &= 2y + 3 \\ f(3, y) &= 2^{y+3} - 3 \\ f(4, y) &\geq 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \}y - \text{mal} \end{aligned}$$

2.3.4 Der μ-Operator

Definition. Ein n-stelliges Prädikat $P(x_1, \ldots, x_n)$ $(n \geq 1)$ ist eine n-stellige Beziehung zwischen natürlichen Zahlen. $P(x_1, \ldots, x_n)$ ordnet jeder Belegung der Variablen $x_i (i = 1, \ldots, n)$ einen Wahrheitswert *wahr* (1) oder *falsch* (0) zu,

$$P : \mathbb{N}^n \longrightarrow \{0, 1\}$$

Beispiel.

$$\begin{aligned} P(x) &\equiv \text{„}x \textit{ ist Primzahl}\text{“} \\ A_1(x,y) &\equiv x|y \\ A_2(x,y) &\equiv x > y \end{aligned}$$

Definition. Ein n-stelliges Prädikat $P(x_1,\ldots,x_n)$ heißt primitiv-rekursiv, wenn es eine primitiv-rekursive Funktion $f(x_1,\ldots,x_n)$ gibt, so dass

$$P(x_1,\ldots,x_n) \text{ g.d.w. } f(x_1,\ldots,x_n) = 0$$

Bemerkung.

1. Es können mehrere solcher Funktionen existieren.
2. Da jede primitiv-rekursive Funktion TURING-berechenbar ist, ist jedes primitiv-rekursive Prädikat entscheidbar.

Beschränkter μ-Operator
Sei $P(x_1,\ldots,x_n,y)$ ein $(n+1)$-stelliges Prädikat. Dann sei

$$\mu y|_{y\le z} P(x_1,\ldots,x_n,y) = \begin{cases} \text{das kleinste } y \text{ mit } 0 \le y \le z, \\ \text{so dass } P(x_1,\ldots,x_n,y), \text{ falls ein solches } y \text{ existiert} \\ 0 \quad \text{sonst} \end{cases}$$

Liegt ein $(n+1)$-stelliges Prädikat vor, so kann man mit Hilfe des beschränkten μ-Operators eine $(n+1)$-stellige Funktion definieren durch

$$g(x_1,\ldots,x_n,z) = \mu y|_{y\le z} P(x_1,\ldots,x_n,y)$$

Satz 2.14. Ist $P(x_1,\ldots,x_n,y)$ ein primitiv-rekursives Prädikat, so ist

$$g(x_1,\ldots,x_n,z) = \mu y|_{y\le z} P(x_1,\ldots,x_n,y)$$

primitiv-rekursiv.

Beweis. Nach Voraussetzung gibt es eine primitiv-rekursive Funktion $f(x_1,\ldots,x_n,y)$ mit $f(x_1,\ldots,x_n,y) = 0 \Longleftrightarrow P(x_1,\ldots,x_n,y)$.
Man hat

$$\begin{aligned} g(x_1,\ldots,x_n,0) &= 0 \\ g(x_1,\ldots,x_n,z+1) &= g(x_1,\ldots,x_n,z) + (z+1)\ \overline{sg}\left(f(x_1,\ldots,x_n,z+1)\ sg\left(\prod_{i=0}^{z} f(x_1,\ldots,x_n,i\right)\right) \end{aligned}$$

1. Fall: Für ein $y \le z$ sei $f(x_1,\ldots,x_n,y) = 0$.
Dann ist der Funktionswert $g(x_1,\ldots,x_n,z)$ und das steht dann auch rechts.

2. Fall: Für alle $y \le z$ sei $f(x_1,\ldots,x_n,y) \neq 0$.
Dann ist der Funktionswert

- $z+1$, wenn $f(x_1,\ldots,x_n,z+1) = 0$
- 0, wenn $f(x_1,\ldots,x_n,z+1) \neq 0$

Das steht dann auch rechts, weil $g(x_1, \ldots, x_n, z) = 0$ usw.

Die rechts vorkommenden Funktionen sind primitiv-rekursiv □

Beispiel.

$$x \cdot /y = \begin{cases} [x/y] & \text{für } y \neq 0 \\ 0 & \text{für } y = 0 \end{cases}$$

Für $y \neq 0$ ist $x \cdot /y$ die größte Zahl t mit $t \leq x/y$,
d.h. $x \cdot /y$ ist die größte ganze Zahl t mit $t \cdot y \leq x$,
d.h. $x \cdot /y$ ist die kleinste ganze Zahl t mit $(t+1) \cdot y > x$,
also $x \cdot /y = \mu t|_{t \leq x} [y(t+1) > x]$.
Stimmt auch für $y = 0$. („>“ - Prädikat ist primitiv-rekursiv)

Unbeschränkter μ-Operator
Weglasssen der Beschränkung für die vom μ-Operator betroffene Variable:
$P(x_1, \ldots, x_n, y)$ sei ein $(n+1)$-stelliges Prädikat, das der Bedingung

$$\boxed{\forall x_1 \, \forall x_2 \, \ldots \forall x_n \, \exists y \; P(x_1, \ldots, x_n, y)}$$

genügt.

$$\mu y P(x_1, \ldots, x_n, y) = \text{kleinstes } y, \text{ für das } P(x_1, \ldots, x_n, y) \text{ erfüllt ist}$$

Definition einer n-stelligen Funktion mittels unbeschränktem μ-Operator:

$$g(x_1, \ldots, x_n) = \mu y P(x_1, \ldots, x_n, y)$$

Satz 2.15. Ist $P(x_1, \ldots, x_n, y)$ ein primitiv-rekursives Prädikat,
so ist $g(x_1, \ldots, x_n) = \mu y P(x_1, \ldots, x_n, y)$ TURING-berechenbar.

Beweis. Es gibt nach Voraussetzung eine primitv-rekursive Funktion f mit

$$f(x_1, \ldots, x_n, y) = 0 \;\; \text{g.d.w.} \;\; P(x_1, \ldots, x_n, y)$$

Für vorgegebene $x_1, \ldots, x_n$ gibt es nach Voraussetzung ein y mit $f(x_1, \ldots, x_n, y) = 0$.
Man berechne mit einer TM $f(x_1, \ldots, x_n, 0), \ldots$ bis Funktionswert $= 0$. Dann ist $g(x_1, \ldots, x_n)$ berechnet. □

Eine mittels unbeschränktem μ-Operator aus einer primitiv-rekursiven Funktion gebildete Funktion ist i.A. nicht wieder primitiv-rekursiv.
Man kann zeigen, dass die ACKERMANN-Funktion auf diese Weise gebildet werden kann.

Definition. Eine Funktion heißt μ-rekursiv, wenn sie

(1) eine der Ausgangsfunktionen U_n^i, C_n^q, N ist

(2) aus diesen Ausgangsfunktionen durch endlich viele Anwendungen des Substitutions- und/oder des Induktionsprozesses und/oder des μ-Operators (mit Existenzbedingung) erhalten werden kann.

Satz 2.16. Jede μ-rekursive Funktion ist TURING-berechenbar.

Beweis. Es genügt zu bemerken, dass im Beweis von Satz 2.15 das f nur TURING-berechenbar vorausgesetzt werden muss. □

Lässt man für das Prädikat $P(x_1, \ldots, x_n, y)$ die Bedingung $\forall x_1 \ldots \forall x_n \; \exists y \; P(x_1, \ldots, x_n, y)$ weg, so wird durch den μ-Operator i.A. eine partielle Funktion definiert.
Eine Definition wie oben liefert dann die Klasse der partiellen μ-rekursiven Funktionen.
Dann gilt auch

Satz 2.17. Jede partielle μ-rekursive Funktion ist (partiell) TURING-berechenbar.

2.3.5 `loop`- und `while`-Berechenbarkeit

`loop`-Programme

`loop`-Programme bestehen aus folgenden syntaktischen Komponenten:

- abzählbar unendlich viele Variable: $x_0, x_1, \ldots$
- Konstanten: $0, 1, \ldots$
- Trennsymbole: $:=$, ;
- Operationssymbole: $+, \dot{-}$
- Schlüsselwörter: `loop`, `do`, `end`

Induktive Definition der Syntax:

1. Wertzuweisungen der Form

   $$x_i := c \text{ und } x_i := x_i + c \text{ und } x_i := x_i \dot{-} c$$

   mit einer Konstanten c sind `loop`-Programme.

2. Sind P_1 und P_2 `loop`-Programme, so ist $P_1; P_2$ ein `loop`-Programm.

3. Falls P ein `loop`-Programm ist, so ist

   $$\texttt{loop } x_i \texttt{ do } P \texttt{ end}$$

   ein `loop`-Programm.

Die Semantik von `loop`-Programmen ist die übliche:
Wertzuweisung wie üblich, $\dot{-}$ ist die arithmetische Differenz, $P_1; P_2$ ist die Sequenz von P_1 und P_2.
`loop` x_i `do` P `end` bedeutet die Ausführung von P so oft, wie der Wert von x_i zu Beginn angibt. (also Zählschleife)

Definition. Eine Funktion $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ heißt `loop`-berechenbar, wenn es ein `loop`-Programm gibt, das gestattet mit jeder Variablenbelegung $n_1, \ldots, n_k$ für (etwa) die Variablen $x_1, \ldots, x_k$ und sonst beliebig hält mit der Variablenbelegung $f(n_1, \ldots, n_k)$ für eine Variable (etwa x_0) und sonst beliebig.

Beispiel.

a) Nachfolgerfunktion: $x_0 := x_1 + 1$

b) Summe: $x_0 := x_1$; `loop` x_2 `do` $x_0 := x_0 + 1$ `end` abgekürzt: $x_0 := x_1 + x_2$

c) Produkt $x_0 := 0$; `loop` x_2 `do` $x_0 := x_0 + x_1$ `end`

Welches sind die `loop`-berechenbaren Funktionen?

Satz 2.18. f ist genau dann `loop`-berechenbar, wenn f primitiv-rekursiv ist.

`while`-Programme

Zusätzlich noch

4. Ist P ein `while`-Programm, so auch

$$\texttt{while } x_i \neq 0 \texttt{ do } P \texttt{ end}$$

mit der Semantik: P wird solange ausgeführt, wie $x_i \neq 0$ ist.
Man kann dann auf die `loop`-Konstruktion verzichten:
Für `loop` x `do` P `end` mit einer neuen Variable y:

$$y := x;\ \texttt{while } y \neq 0 \texttt{ do } y := y - 1;\ P \texttt{ end}$$

`while`-berechenbare Funktionen werden analog zu oben definiert.
`while`-berechenbare Funktionen sind i.A. partiell, weil es unendliche Schleifen geben kann.
Es gilt nun

Satz 2.19. Für eine Funktion $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ sind äquivalent

(1) f ist TURING-berechenbar

(2) f ist μ-rekursiv

(3) f ist `while`-berechenbar

Bemerkung. Die Aussage von Satz 2.19 bleibt erhalten, wenn f partiell ist.

Alle diese Berechenbarkeitsbegriffe führen auf dieselbe Funktionenklasse!
Dies gibt Anlass zur sogenannten

CHURCHschen These
Jede (im intuitiven Sinne) berechenbare Funktion ist μ-rekursiv (d.h. TURING-berechenbar, ...) .
Diese These ist (natürlich) nicht beweisbar, denn sie bezieht sich auf intuitiv berechenbare Funktionen. (Sie ist eventuell widerlegbar.)
Aber jede Präzisierung führte bisher auf obige Funktionenklassen.
Jede der Präzisierungen ist ein Explikat des Algorithmusbegriffes.

2.4 Komplexität von Algorithmen

Man möchte die „*Güte*“ von Algorithmen messen. Es gibt viele Gütekriterien, z.B. Rechenzeit und Speicherplatz.
Wir wählen als Rechenmodell die TM und zwar eine TM mit k zweiseitig unendlichen Bändern.

2.4.1 Grundlagen

Definition (*Zeitkomplexität*)**.** Wenn für ein Inputwort der Länge n die TM M höchstens $T(n)$ Schritte macht, bevor sie hält, so heißt M eine $T(n)$-beschränkte TM oder eine TM der Zeitkomplexität $T(n)$. Die von M akzeptierte Sprache heißt dann auch von der Zeitkomplexität $T(n)$.

Definition (*Speicherkomplexität*)**.** Wenn M für jedes Inputwort der Länge n auf jedem Band höchstens $S(n)$ Zellen (Felder) examiniert, so heißt M von der Speicherkomplexität $S(n)$. Die von M akzeptierte Sprache heißt dann ebenfalls von der Speicherkomplexität $S(n)$.

Beispiel. Wir betrachten die Sprache $L = \{wcw^R : \ w \in \{0,1\}^*\}$.
Es gibt eine TM M mit zwei Bändern, die das Input links von c auf Band 2 kopiert. Wenn c gefunden wird, wird die 2. LSV im gerade kopierten Wort nach links und die 1. LSV gleichzeitig im Input weiter nach rechts bewegt und die Zeichen unter den LSVs werden verglichen. Wenn alle Symbole gleich sind und die Strings gleich lang so wird akzeptiert.
Offenbar macht M höchstens $n+1$ Schritte für einen Input der Länge n.
Daher hat M (und also auch L) die Zeitkomplexität $n+1$. M (und L) hat auch die Speicherkomplexität $n+1$.

Definition. Eine NTM M hat die Zeitkomplexität $T(n)$, wenn sie für jedes Inputwort der Länge n bei jeder Folge von Wahlmöglichkeiten höchstens $T(n)$ Schritte macht.
M hat die Speicherkomplexität $S(n)$, wenn bei jeder Folge von Wahlmöglichkeiten M auf jedem Band höchstens $S(n)$ Zellen examiniert bei einem Inputwort der Länge n. Entsprechend für die akzeptierten Sprachen.

Definition (*Komplexitätsklassen*)**.** Die Familie aller Sprachen der Zeitkomplexität $T(n)$ wird mit DTIME($T(n)$) bezeichnet. Die Familie aller Sprachen nichtdeterministischer Zeitkomplexität $T(n)$ wird mit NTIME($T(n)$) bezeichnet. Analog für die DSPACE($S(n)$) und NSPACE($S(n)$).

Beispiel. $L = \{wcw^R : w \in \{0,1\}^*\} \in$ DTIME$(n+1)$ und damit natürlich auch
$\in$ NTIME$(n+1)$.

Im folgenden betrachten wir nur Zeitkomplexität. Ein wesentlicher Teil der Komplexitätstheorie befasst sich mit den Beziehungen zwischen Komplexitätsklassen.
Beispiele sind die folgenden Sätze:

Satz 2.20. Ist $L \in$ DTIME($T(n)$), so wird L akzeptiert in $T^2(n)$ Zeit von einer TM mit einem Band.

Satz 2.21. Ist $L \in$ NTIME $(f(n))$, so gibt es eine (von L abhängige) Konstante c, so dass $L \in$ DTIME$(c^{f(n)})$ liegt.

2.4.2 Praktisch unlösbare Probleme

Die Klassen $\mathcal{P}$ und $\mathcal{NP}$

Frage. Wann ist ein Problem effizient (bzgl. Zeit) lösbar?

Naheliegend: Wenn es in einer Zeit lösbar ist, die *polynomial* von der Länge des Inputs abhängt.
Zwar scheint zweifelhaft, dass ein Algorithmus der Komplexität n^{100} effizient ist, doch stellt sich heraus, dass die meisten praktischen Probleme mit polynomialer Komplexität gelöst werden können in Zeiten mit sehr niedrigen Polynomgraden.

Wir definieren daher $\mathcal{P}$ als Klasse aller Sprachen, die durch DTM mit polynomialer Zeitkomplexität akzeptiert werden, d.h.

$$\mathcal{P} = \bigcup_{p \text{ Polynom}} \text{DTIME}(p(n))$$

Analog definiert man

$$\mathcal{NP} = \bigcup_{p \text{ Polynom}} \text{NTIME}(p(n))$$

Trivialerweise gilt $\mathcal{P} \subseteq \mathcal{NP}$.
Die Frage $\mathcal{P} = \mathcal{NP}$ ist offen und gilt als das schwierigste Problem der Informatik.

Definition. Die Sprache L' heißt reduzierbar auf die Sprache L in polynomialer Zeit, $L' \leq L$, wenn es eine (deterministische) TM M mit der Zeitkomplexität $T(n)$ gibt, so dass $T(n)$ ein Polynom in n ist und M erzeugt aus dem Input x das Output y, so dass $x \in L'$ g.d.w. $y \in L$. (Der Test $x \in L'$ kann reduziert werden auf den Test $y \in L$.)

Lemma (1)**.** Sei $L' \leq L$. Dann gilt

(a) $L \in \mathcal{P} \Longrightarrow L' \in \mathcal{P}$

(b) $L \in \mathcal{NP} \Longrightarrow L' \in \mathcal{NP}$

Beweis. (von (a) analog). Werde also L akzeptiert in der Zeit $p_2(n)$ und sei L' in der Zeit $p_1(n)$ reduzierbar durch $M = (\Sigma, \ldots)$ auf L, wobei p_1 und p_2 Polynome in n sind.
Sei dann $x \in \Sigma^*$ mit $|x| = n$ und y das Output der Reduktion mit Input x. Dann ist $|y| \leq p_1(n)$ (da höchstens ein Symbol/Schritt ausgegeben werden kann).
Um nun zu testen, ob $y \in L$, sind höchstens $p_2(p_1(n))$ Schritte nötig. Insgesamt höchstens $p_1(n) + p_2(p_1(n))$ Schritte!
Aber $p_1(n) + p_2(p_1(n))$ ist Polynom in n $\Longrightarrow$ Behauptung. □

Lemma (2)**.** Die Hintereinanderausführung (Komposition) von 2 Reduktionen in polynomialer Zeit ist wieder eine solche.

Beweis. analog.

$$(L_1 \leq L_2 \leq L_3 \Longrightarrow L_1 \leq L_3)$$

□

Primzahlproblem: Ist N eine Primzahl?

Algorithmus: Teste sukzessive, ob $2, 3, \ldots, [\sqrt{N}]$ Teiler von N sind.
Dieser Algorithmus hat Zeitkomplexität $O(\sqrt{N} \cdot \log^2 N)$.
Ist er polynomial? (sublinear?) Das wäre so, wenn das Input N unär codiert würde. Man kann aber z.B. N binär codieren, dann ist die Länge des Inputs $n = \log_2 N$.
Komplexität des Algorithmus bezogen auf die Inputlänge n größer als

$$\sqrt{N} = 2^{\frac{1}{2}\log_2 N} = 2^{\frac{n}{2}},$$

also exponentiell!
Für das Primzahlproblem PRIME gilt PRIME $\in \mathcal{P}$ (bekannt erst seit 2002).
(Komplexität $O(n^{12})$)
Praktisch werden *problematische* Algorithmen (Monte-Carlo-Algorithmen) benutzt, die sehr schnell sind und die Form haben:

Input: $N \geq 3$, ungerade

1. wähle $a \in \{2, \ldots, N-2\}$ zufällig
2. mache mit a „*verschiedene Rechnungen*“ die auch sehr schnell sind
3. in Abhängigkeit vom Ergebnis Output „*wahrscheinlich prim*“ oder „*zusammengesetzt*“

Der Algoritmus hat die Eigenschaft:

1. wenn Output „*zusammengesetzt*“, so ist N zusammengesetzt
2. wenn N nicht prim, so ist die Wahrscheinlichkeit, dass Output „*wahrscheinlich prim*“ $\leq \frac{1}{2}$

Man könnte das Problem $\mathcal{P} = \mathcal{NP}$ so angehen, indem man nach einem „*schwersten*“ Problem aus $\mathcal{NP}$ fragt.
„*Schwerstes*“ Problem in dem Sinne, dass sich **jedes** andere Probem aus $\mathcal{NP}$ „*leicht*“ auf dieses „*schwerste*“ Problem reduzieren lässt.
„*Leicht*“ wäre hier offenbar mit „*in polynomialer Zeit*“ gleichzusetzen.
Das führt auf:

Definition. Eine Sprache L heißt $\mathcal{NP}$-vollständig, wenn $L \in \mathcal{NP}$ und jede Sprache aus $\mathcal{NP}$ in polynomialer Zeit auf L reduzierbar ist.
Eine Sprache L heißt $\mathcal{NP}$-schwer ($\mathcal{NP}$-hart), wenn jede Sprache aus $\mathcal{NP}$ in polynomialer Zeit auf L reduzierbar ist.

Bemerkung.

1. Hat man eine $\mathcal{NP}$-vollständige Sprache L, so ist $\mathcal{P} = \mathcal{NP}$ gleichbedeutend mit $L \in \mathcal{P}$.

   Denn: Wenn $\mathcal{P} = \mathcal{NP}$, so wegen $L \in \mathcal{NP} \Longrightarrow L \in \mathcal{P}$.
   Wenn $L \in \mathcal{P}$, so sei $L' \in \mathcal{NP} \overset{L-\mathcal{NP}-\text{vollst.}}{\Longrightarrow} L' \leq L$.
   $\overset{\text{L.1}}{\Longrightarrow} L' \in \mathcal{P}$, also $\mathcal{P} = \mathcal{NP}$

2. Hat man eine $\mathcal{NP}$-vollständige Sprache L_0, so kann man die $\mathcal{NP}$-Vollständigkeit einer anderen Sprache $L_1 \in \mathcal{NP}$ dadurch beweisen, dass man zeigt, dass $L_0 \leq L_1$ ist.

   Denn: Zu zeigen, dass ein beliebiges $L \in \mathcal{NP}$ auf L_1 reduzierbar ist.
   Aber, da L_0 $\mathcal{NP}$-vollständig $\Longrightarrow \underline{L \leq L_0} \Longrightarrow L \leq L_1$

Einige $\mathcal{NP}$-vollständige Probleme
Um für eine erste Sprache $\mathcal{NP}$-Vollständigkeit zu zeigen, muss man nachweisen, dass jede Sprache aus $\mathcal{NP}$ in polynomialer Zeit auf diese reduziert werden kann:

Erfüllbarkeitsproblem für BOOLsche Ausdrücke
B.A. (aussagenlogische Formeln) werden gebildet aus Variablen, Klammern und den logischen Operatoren $\wedge$ (log. und), $\vee$ (log. oder), $\neg$ (Negation) mit der Rangfolge $\neg$, $\wedge$, $\vee$.
Die Variablen können die Werte 0 (`false`) und 1 (`true`) haben. Ein B.A. hat dann auch - in Abhängigkeit von den Werten der Variablen - ebenfalls den Wert 0 bzw. 1 - nach den *üblichen* Regeln.
Ein B.A. heißt erfüllbar, wenn es eine Belegung der Variablen gibt, so dass der B.A. den Wert 1 hat.

Z.B. ist $x_1 \wedge x_2$ erfüllbar (Belegung: $x_1 = 1, x_2 = 1$)
Z.B. ist $x_1 \wedge \neg x_1$ nicht erfüllbar

Das *Erfüllbarkeitsproblem* (SAT) ist nun:

Ist der B.A. E erfüllbar?

Wir stellen das Problem mittels einer Sprache L_{erf} wie folgt da:
Seien die Variablen des Ausdruckes $x_1, \ldots, x_k$.
x_i sei codiert durch das Symbol x gefolgt von der binären Darstellung von i, z.B. $x_3 : x11$
Das Alphabet von L_{erf} ist dann

$$\{\vee, \wedge, \neg, (,), x, 0, 1\}$$

L_{erf} ist die Menge aller codierten erfüllbaren B.A.
Es gibt nun:

Satz von Cook
Das Erfüllbarkeitsproblem ist $\mathcal{NP}$-vollständig.
Man zeigt

a) $L_{erf} \in \mathcal{NP}$

b) Für $L \in \mathcal{NP}$ gilt $L \leq L_{erf}$

zu a) *Guess-and-Check-Methode*: Wähle *nichtdeterministisch* eine Belegung der Variablen im B.A. und überprüfe dann, ob diese Belegung den B.A. wahr macht.
Das geht in polynomialer Zeit.

Weitere $\mathcal{NP}$-vollständige Probleme sind:

Erfüllbarkeitsproblem für B.A. in 3KNF - L_{3erf} (3-SAT): B.A. E in KNF, wenn E Konjunktion von Disjunktionen von Literalen (Variablen bzw. negierte Variaben), z.B.

$$E = (x_1 \vee x_2 \vee \neg x_3 \vee \neg x_1) \wedge (x_2 \vee \neg x_2 \vee x_3) \wedge \ldots$$

Ein 3KNF: Ein KNF und jede Disjunktion hat genau 3 Literale

Hamilton-Kreis-Problem: Gibt es im ungerichteten Graphen $G = (K, R)$ einen Hamilton-Kreis. d.h. einen Kreis, der durch jeden Knoten genau einmal hindurchgeht?

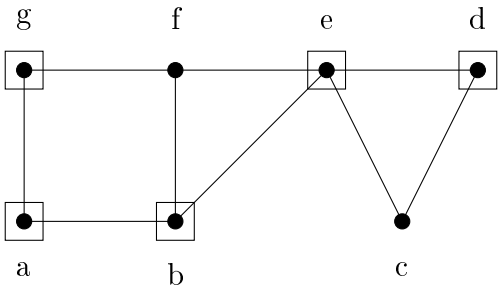
Problem des Handlungsreisenden: Gegeben ist ein ungerichteter vollständiger Graph G, dessen Knoten mit natürlichen Zahlen gewichtet sind, und eine natürliche Zahl l.
Gibt es eine Rundreise, d.h. einen Hamilton-Kreis des Gesamtgewichtes $\leq l$?

Die Beweise erfolgen immer gemäß Bem. 2, d.h. man zeigt für jede fragliche Sprache L

a) $L \in \mathcal{NP}$

b) Man wählt eine $\mathcal{NP}$-vollständige Sprache L' und zeigt $L' \leq L$.

Beispiel: Knotenhüllenproblem:
Gegeben sei ein gerichteter Graph $G = (K, R)$ (Knotenmenge K, Kantenmenge R).
Eine Teilmenge $A \subseteq K$ heißt Knotenhülle von G, wenn für jede Kante $(v, w) \in R$ v oder w in A liegt.



$$A = \{a, b, d, e, g\} \text{ ist Knotenhülle}$$
$$A' = \{b, d, e, g\} \text{ ist Knotenhülle}$$

Das Knotenhüllenproblem:
Gegeben ist ein Graph G und eine natürliche Zahl $\leq k$.
Besitzt G eine Knotenhülle der Größe $\leq k$?
Darstellung als Sprache:
k binär, dann Liste der Knoten k_i, codiert, z.B. k_2 als k_{10}, dann Liste der Kanten (k_i, k_j), codiert, z.B. (k_2, k_3) als (k_{10}, k_{11}).
L_{kk} besteht aus allen solchen Strings, die k und G repräsentieren, so dass G eine Knotenhülle der Größe $\leq k$ besitzt.

Satz 2.22. L_{kk} ist $\mathcal{NP}$-vollständig.

Beweis.

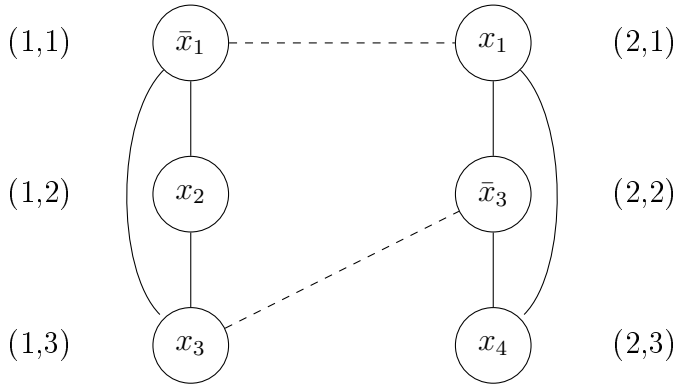
a) $L_{kk} \in \mathcal{NP}$: Wähle nichtdeterministisch k Knoten und teste ob sie eine Hülle bilden. (Geht in polynomialer Zeit.)

b) $L_{3erf} \leq L_{kk}$: Sei $E = E_1 \wedge E_2 \wedge \ldots \wedge E_q$ ein B.A. in 3KNF mit $E_i = (\alpha_{i1} \vee \alpha_{i2} \vee \alpha_{i3})$, α_{ij} Literale.
Wir konstruieren einen ungerichteten Graphen $G = (K, R)$:

$$K = \{(i, j) : 1 \leq i \leq q, 1 \leq j \leq 3\}$$

□

Beispiel. $E = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3 \vee x_4)$



$$R = R_1 \cup R_2 = \{((i,j),(i,k)) : j \neq k\} \cup \{((i,j),(k,l)) : i \neq k, \alpha_{ij} = \neg\alpha_{kl}\}$$

Behauptung. G besitzt eine Knotenhülle der Größe $\leq 2q$ g.d.w. E ist erfüllbar.

Beweis.

a) E sei erfüllbar. Dann gibt es eine Belegung, so dass für jedes i ein $\alpha_{i\nu_i}$ den Wert 1 hat. Man streiche die q Knoten (i,ν_i) aus K. Die verbleibenden $2q$ Knoten bilden eine Überdeckung A von G: Sei $(v,w) \in R$.
Wenn $(v,w) \in R_1$, so ist $v \in A$ oder $w \in A$, da nur einer der Knoten $(i,1),(i,2),(i,3)$ fehlt!
Wenn $(v,w) \notin A$, so $\alpha_{ij} = 1$ (bei der Belegung) und also $\alpha_{kl} = 0$, also $(k,l) \in A$.

b) G habe eine Knotenhülle A der Größe $\leq 2q$. Für jedes i enthält A dann genau 2 Knoten der Form (i,j).
D.h. für jedes i „*fehlt in* A“ genau ein Knoten (i,ν_i). Dann sei $\alpha_{i\nu_i}$ gerade mit 1 belegt. Es entsteht kein Konflikt, denn $\alpha_{i\nu_i} = \neg\alpha_{j\nu_j}$ für $i \neq j$ ist ausgeschlossen, weil dann $((i,\nu_i),(j,\nu_j)) \in R_2$ und keiner der Knoten in A. $\longrightarrow$ ↯

□

Kapitel 3

Mathematische Logik mit Informatikanwendungen

3.1 Aussagenlogik

In der Informatik befasst man sich mit *Elementaraussagen* (Atomen) $A, B \ldots$, die nur entweder den Wert *wahr* (1) oder *falsch* (2) annehmen können und mit der (*logischen*) Verknüpfung von solchen Aussagen durch definierte Operationen zu *Formeln*.

Ausgehend von den Atomen definiert man die *Menge der Formeln FO* wie folgt:

1. Jedes Atom ist einen Formel.

2. Ist $\alpha \in FO$, so ist auch $\neg\alpha \in FO$ (*Negation*).

3. Sind $\alpha, \beta \in FO$, so ist auch $\alpha \vee \beta \in FO$ (*Disjunktion*).

4. Die mittels 1.-3. gebildeten Formeln sind alle Formeln.

Als *Abkürzung* werden noch folgende Zeichen benutzt:
$\wedge$ (*Konjunktion*), $\rightarrow$ (*Implikation*), $\leftrightarrow$ (*Äquivalenz*), und zwar

$\alpha \rightarrow \beta$ für $(\neg\alpha) \vee \beta$
$\alpha \wedge \beta$ für $\neg((\neg\alpha) \vee (\neg\beta))$
$\alpha \leftrightarrow \beta$ für $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$

Die so definierten Formeln sind noch nichts weiter als Zeichenketten, die bestimmten (oben beschriebenen) syntaktischen Regeln genügen - sie stellen gerade die *Syntax* der Aussagenlogik dar.

Für die *Semantik* der Aussagenlogik ordnet man jeder Formel einen Wahrheitswert 1 (wahr) oder 0 (falsch) zu, in Abhängigkeit einer zugrunde gelegten Bewertung der Atome in der Formel:

$$\text{WERT}(\neg\alpha) = \begin{cases} 1 & : \text{WERT}(\alpha) = 0 \\ 0 & : \text{sonst} \end{cases}$$

$$\text{WERT}(\alpha \vee \beta) = \begin{cases} 1 & : \text{WERT}(\alpha) = 1 \text{ oder } \text{WERT}(\beta) = 1 \\ 0 & : \text{sonst} \end{cases}$$

Man hat offenbar

$$\text{WERT}(\alpha \wedge \beta) = \begin{cases} 1 & : \text{WERT}(\alpha) = \text{WERT}(\beta) = 1 \\ 0 & : \text{sonst} \end{cases}$$

$$\text{WERT}(\alpha \to \beta) = \begin{cases} 1 & : \text{WERT}(\alpha) = 0 \text{ oder } \text{WERT}(\beta) = 1 \\ 0 & : \text{sonst} \end{cases}$$

$$WERT(\alpha \leftrightarrow \beta) = \begin{cases} 1 & : \text{WERT}(\alpha) = \text{WERT}(\beta) \\ 0 & : \text{sonst} \end{cases}$$

Beweis. Fallunterscheidung! □

Beispiel. Für die Atome A, B, C sei die Bewertung $\text{WERT}(A) = \text{WERT}(B) = 1$ und $\text{WERT}(C) = 0$. Dann wird

$$\text{WERT}((\neg A \vee (B \vee C)) \wedge ((C \vee B) \wedge \neg C)) = 1$$

Definition. Die Formel α heißt *erfüllbar*, wenn es eine Bewertung gibt mit $\text{WERT}(\alpha) = 1$ (d.h. man kann α durch eine Bewertung der Atome wahr machen).

Beispiel.

$$(A \vee B) \wedge \neg B$$

ist erfüllbar mit $\text{WERT}(A) = 1$ und $\text{WERT}(B) = 0$.

$$(\neg A \wedge (A \vee B)) \wedge \neg B$$

ist nicht erfüllbar.

Definition. Die Formel α heißt *Tautologie* oder *allgemeingültig*, wenn für jede Bewertung (der Atome) $\text{WERT}(\alpha) = 1$ gilt.

Beispiel. $A \vee \neg A$ ist eine Tautologie.

Definition. Die Formeln α und β heißen (*logisch*) *äquivalent*, wenn für jede Bewertung $\text{WERT}(\alpha) = \text{WERT}(\beta)$ gilt (in Zeichen: $\alpha \equiv \beta$).

Satz 3.1.

1. α ist nicht erfüllbar $\iff \neg\alpha$ ist Tautologie
2. α und β sind äquivalent $\iff \alpha \leftrightarrow \beta$ ist Tautologie

Beweis.

1. α ist nicht erfüllbar
   $\iff$ für jede Bewertung gilt $\text{WERT}(\alpha) = 0$
   $\iff$ für jede Bewertung gilt $\text{WERT}(\neg\alpha) = 1$
   $\iff \neg\alpha$ ist Tautologie
2. α und β sind äquivalent

$\iff$ für jede Bewertung gilt $\mathrm{WERT}(\alpha) = \mathrm{WERT}(\beta)$

$\iff$ für jede Bewertung gilt $\mathrm{WERT}(\alpha \leftrightarrow \beta) = 1$

$\iff$ $\alpha \leftrightarrow \beta$ ist Tautologie

□

Man erhält nun leicht folgende aussagenlogische Gesetze (Beweis mit Wahrheitswerttabellen):

doppelte Negation	$\neg\neg\alpha \equiv \alpha$
Idempotenz	$\alpha \vee \alpha \equiv \alpha$ $\alpha \wedge \alpha \equiv \alpha$
Kommutativität	$\alpha \vee \beta \equiv \beta \vee \alpha$ $\alpha \wedge \beta \equiv \beta \wedge \alpha$ $\alpha \leftrightarrow \beta \equiv \beta \leftrightarrow \alpha$
Assoziativität	$(\alpha \vee \beta) \vee \gamma \equiv \alpha \vee (\beta \vee \gamma)$ $(\alpha \wedge \beta) \wedge \gamma \equiv \alpha \wedge (\beta \wedge \gamma)$ $(\alpha \leftrightarrow \beta) \leftrightarrow \gamma \equiv \alpha \leftrightarrow (\beta \leftrightarrow \gamma)$
Distributivität	$\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$ $\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
DE MORGAN	$\neg(\alpha \vee \beta) \equiv (\neg\alpha) \wedge (\neg\beta)$ $\neg(\alpha \wedge \beta) \equiv (\neg\alpha) \vee (\neg\beta)$

Satz 3.2 (Ersetzbarkeitstheorem)**.** Seien α und β äquivalente Formeln ($\alpha, \beta \in FO, \alpha \equiv \beta$). Sei $\gamma \in FO$ eine Formel, in der α (an mindestens einer Stelle) als Teilformel vorkommt. Sei γ' die Formel, die aus γ entsteht, indem irgendein Vorkommen von α durch β ersetzt wird. Dann gilt: $\gamma \equiv \gamma'$.

Beweis. Strukturelle Induktion:

Induktionsanfang: Sei γ atomar. Dann ist $\alpha = \gamma$, also folgt, wegen $\alpha \equiv \beta$, $\gamma \equiv \gamma'$.

Induktionsschritt: Ist $\alpha = \gamma$, dann so wie oben. Sei also $\alpha \neq \gamma$:

1. Sei $\gamma = \neg\gamma_1$ und der Satz gilt für γ_1 (nach Induktionsvoraussetzung), d.h. es ist γ_1' die durch Ersetzung von α durch β aus γ_1 sich ergebende Formel, so gilt $\gamma_1 \equiv \gamma_1'$. Daraus folgt $\neg\gamma_1 \equiv \neg\gamma_1'$, d.h. $\gamma \equiv \gamma'$.
2. Sei $\gamma = \gamma_1 \vee \gamma_2$: α kommt (an der zu ersetzenden Stelle) entweder in γ_1 oder in γ_2 vor. Sei α o.B.d.A. in γ_1. Wird γ_1' durch Ersetzung von α durch β gebildet, so gilt nach Induktionsvoraussetzung $\gamma_1 \equiv \gamma_1'$. Hieraus folgt $\gamma_1 \vee \gamma_2 \equiv \gamma_1' \vee \gamma_2$, d.h. $\gamma \equiv \gamma'$.

□

Mit Hilfe des Ersetzbarkeitstheorems kann man nun Äquivalenzbeweise auch anders führen:

$$(A \vee (B \vee C)) \wedge (C \vee \neg A) \equiv (B \wedge \neg A) \vee C,$$

denn es gilt:

$$\begin{aligned}
(A \vee (B \vee C)) \wedge (C \vee \neg A) &\equiv ((A \vee B) \vee C) \wedge (C \vee \neg A) && \text{Assoziativität}\\
&\equiv (C \vee (A \vee B)) \wedge (C \vee \neg A) && \text{Kommutativität}\\
&\equiv C \vee ((A \vee B) \wedge \neg A) && \text{Distributivität}\\
&\equiv C \vee (\neg A \wedge (A \vee B)) && \text{Kommutativität}\\
&\equiv C \vee ((\neg A \wedge A) \vee (\neg A \wedge B)) && \text{Distributivität}\\
&\equiv C \vee (\neg A \wedge B) && \text{wegen: } (\neg\alpha \wedge \alpha) \vee \beta \equiv \beta\\
&\equiv C \vee (B \wedge \neg A) && \text{Kommutativität}\\
&\equiv (B \wedge \neg A) \vee C && \text{Kommutativität}
\end{aligned}$$

$(\neg\alpha \wedge \alpha) \vee \beta$ gilt offensichtlich, da $\text{WERT}(\neg\alpha \wedge \alpha) = 0$ ist für alle Belegungen ($\neg\alpha \wedge \alpha$ ist nicht erfüllbar).

Semantischer Folgerungsbegriff

Definition. Die Formel β folgt aus einer Menge $\alpha_1, \ldots, \alpha_n$ von Formeln, wenn für alle Bewertungen mit $\text{WERT}(\alpha_1) = \cdots = \text{WERT}(\alpha_n) = 1$ auch $\text{WERT}(\beta) = 1$ gilt. In Zeichen: $\alpha_1, \ldots, \alpha_n \models \beta$

Satz 3.3. Folgende Aussagen sind äquivalent:

(i) $\alpha_1, \ldots, \alpha_n \models \beta$

(ii) $(\alpha_1 \wedge \cdots \wedge \alpha_n) \rightarrow \beta$ ist Tautologie

(iii) $\alpha_1 \wedge \cdots \wedge \alpha_n \wedge \neg\beta$ ist nicht erfüllbar.

Beweis.

(i)⇒(ii) Sei $\alpha_1, \ldots, \alpha_n \models \beta$, d.h. für alle Bewertungen mit $\text{WERT}(\alpha_1), \ldots, \text{WERT}(\alpha_n) = 1 \Rightarrow \text{WERT}(\beta) = 1$. Aber $(\alpha_1 \wedge \cdots \wedge \alpha_n) \rightarrow \beta = \neg(\alpha_1 \wedge \cdots \wedge \alpha_n) \vee \beta \equiv \neg\alpha_1 \vee \cdots \vee \neg\alpha_n \vee \beta$ ist dann Tautologie.

(ii)⇒(iii) Wenn $(\alpha_1 \wedge \cdots \wedge \alpha_n) \rightarrow \beta \equiv \neg\alpha_1 \vee \cdots \vee \neg\alpha_n \vee \beta$ Tautologie ist, so ist $\neg(\neg\alpha_1 \vee \cdots \vee \neg\alpha_n \vee \beta) \equiv \alpha_1 \wedge \cdots \wedge \alpha_n \wedge \neg\beta$ nicht erfüllbar.

(iii)⇒(i) Wenn $\alpha_1 \wedge \cdots \wedge \alpha_n \wedge \neg\beta$ nicht erfüllbar ist, so folgt für beliebige Bewertungen mit $\text{WERT}(\alpha_1) = \cdots = \text{WERT}(\alpha_n) = 1 \Rightarrow \text{WERT}(\beta) = 1$.

□

Bemerkung. Die Tautologien sind also Formeln, die man ohne Voraussetzung folgern kann.

Beispiel.

$$\begin{aligned}
A \wedge (A \rightarrow B) \models B &\equiv A \wedge (\neg A \vee B) \wedge \neg B\\
&\equiv ((A \wedge \neg A) \vee (A \wedge B)) \wedge \neg B\\
&\equiv (A \wedge B) \wedge \neg B\\
&\equiv A \wedge (B \wedge \neg B)
\end{aligned}$$

ist nicht erfüllbar.

Normalformen Für praktische Anwendungen sind Formeln einer bestimmten Gestalt sinnvoll → *Normalformen*.

Definition. Ein *Literal* ist ein Atom oder ein negiertes Atom. Eine *Klausel* ist eine Disjunktion von Literalen. Die Formel α ist in *konjunktiver Normalform (KNF)*, wenn α eine Konjunktion von Klauseln ist.

Beispiel. $(A \vee B \vee C) \wedge (\neg A \vee C) \wedge (D \vee \neg C \vee B)$ ist in KNF.

Algorithmus KNF

1. Eliminiere $\leftrightarrow$: Ersetze $\alpha \leftrightarrow \beta$ durch $(\alpha \to \beta) \wedge (\beta \to \alpha)$
2. Eliminiere $\to$: Ersetze $\alpha \to \beta$ durch $\neg\alpha \vee \beta$
3. Ziehe Negationen nach innen:

Ersetze $\neg(\alpha \vee \beta)$ durch $\neg\alpha \wedge \neg\beta$

Ersetze $\neg(\alpha \wedge \beta)$ durch $\neg\alpha \vee \neg\beta$

Ersetze $\neg(\neg\alpha)$ durch α

4. Wende Distributivgesetz an: Ersetze $(\alpha \wedge \beta) \vee \gamma$ durch $(\alpha \vee \gamma) \wedge (\beta \vee \gamma)$.

Wegen des Ersetzbarkeitstheorems liefert dieser Algorithmus wirklich eine äquivalente Formel.

Bemerkung. Es gibt eine duale Normalform DNF.

Frage. Wie kann man herausbekommen, ob $\alpha_1, \ldots, \alpha_n \vDash \beta$ gilt?

Nach dem obigen Satz kann man $\alpha_1 \wedge \cdots \wedge \alpha_n \wedge \neg\beta$ auf Nichterfüllbarkeit testen. Das geht etwa mit Tabellen (d.h. Durchprobieren aller Bewertungen), ist aber exponentiell. (Da das Problem - wie wir wissen - $\mathcal{NP}$-vollständig ist, können wir allerdings auf keinen „*schnellen*“ Algorithmus hoffen).

Eine andere Möglichkeit ist die *Resolution*. Dies ist besonders deshalb vorteilhaft, weil es syntaktisch abläuft.

Von einer Formel α ausgehend, bilde man eine äquivalente Formel α' in KNF und bilde aus den dort vorkommenden Klauseln *Klauselmengen*.

$$\alpha' = (L_{1,1} \vee \cdots \vee L_{1,n_1}) \wedge \cdots \wedge (L_{k,1} \vee \cdots \vee L_{k,n_k})$$

liefert die Menge

$$\{\{L_{1,1}, \ldots, L_{1,n_1}\}, \ldots, \{L_{k,1}, \ldots, L_{k,n_k}\}\}.$$

Man beachte, dass dabei gleiche Literale aus einer Klausel in der Klauselmenge nur einmal gezählt werden!

Beispiel. $(A \vee \neg B) \wedge C$ liefert $\{\{A, \neg B\}, \{C\}\}$.

Definition. Eine *Resolvente* zweier Klauseln der Form $K_1 = \{\ldots, A, \ldots\}$ und $K_2 = \{\ldots, \neg A, \ldots\}$ (A Atom), ist die Klausel $(K_1 \backslash \{A\}) \cup (K_2 \backslash \{\neg A\})$.

Beispiel. Eine Resolvente von $\{C, \neg D, A\}, \{D, \neg A\}$ ist $\{C, A, \neg A\}$. Eine andere ist $\{C, \neg D, D\}$.

Definition. Ist Φ eine Klauselmenge, so ist $\mathrm{Res}(\Phi)$ definiert als

$$\mathrm{Res}(\Phi) = \Phi \cup \{R : R \text{ ist Resolvente zweier Klauseln aus } \Phi\}.$$

Definition.

$$\begin{aligned} \mathrm{Res}^0(\Phi) &= \Phi \\ \mathrm{Res}^{n+1}(\Phi) &= \mathrm{Res}(\mathrm{Res}^n(\Phi)) \\ \mathrm{Res}^*(\Phi) &= \bigcup_{n>1} \mathrm{Res}^n(\Phi) \end{aligned}$$

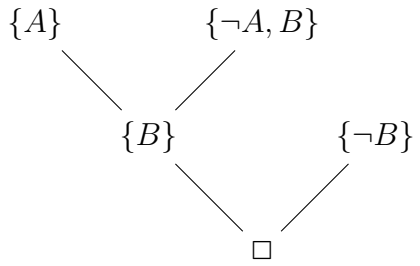
$\square$ bezeichnet die *leere Klausel* (die also kein Literal enthält).

Satz 3.4 (Resolutionsatz der Aussagenlogik)**.** Sei Φ die zu α gehörige Klauselmenge. Dann gilt: α ist nicht erfüllbar $\iff \square \in \mathrm{Res}^*(\Phi)$.

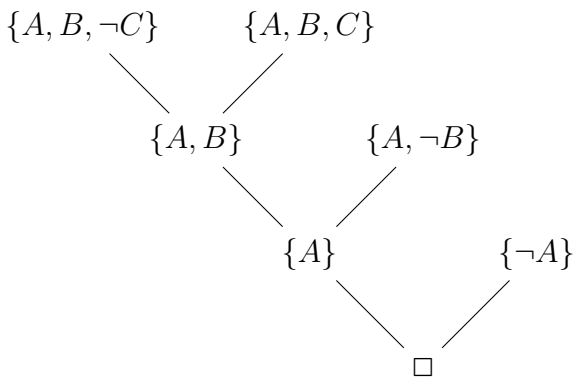
Bemerkung. Man beachte, dass $\alpha_1 \wedge \cdots \wedge \alpha_n \wedge \neg\beta$ nicht erfüllbar ist $\iff \alpha_1, \ldots, \alpha_n \vDash \beta$.

Beispiel.

a) $A \wedge (A \to B) \vDash B$? (siehe oben)
Also $A \wedge (\neg A \vee B) \wedge \neg B \Longrightarrow \Phi = \{\{A\}, \{\neg B\}, \{\neg A, B\}\}$



b) Ist $(A \vee B \vee \neg C) \wedge (\neg A) \wedge (A \vee B \vee C) \wedge (A \vee \neg B)$ unerfüllbar? Ja!



3.2 Prädikatenlogik

Beispiel. $\exists x\ (3 * x > 0)$ ist logischer Ausdruck (Formel) mit Variable (x), die „*quantifiziert*“ sind ($\exists$), mit Funktion ($*$) und Prädikat ($>$).

Notwendigkeit die Sprache zu *erweitern*.

Syntax

Definition. Sei F eine (abzählbare) Menge von *Funktionssymbolen*, P eine (abzählbare) Menge von *Prädikatensymbolen*, dann heißt $BA = (F, P)$ *syntaktische Basis*.
Für jedes $n \in \mathbb{N}$ seien F^n bzw. P^n die Mengen der n-stelligen Funktionssymbole bzw. n-stelligen Prädikatensymbole, so dass gilt

$$F = \bigcup_{n=0}^{\infty} F^n \quad , \quad P = \bigcup_{n=0}^{\infty} P^n$$

V sei abzählbar unendliche Menge von Variablen.

Jetzt definieren wir *Terme* und *Formeln*:

Definition. Die Menge der Terme T (bezüglich einer Basis BA) ist die kleinste Menge, für die gilt:

(1) $V \subseteq T$ (d.h. jede Variable ist ein Term)

(2) $t_1, \ldots, t_n \in T, f \in F^n \Longrightarrow f(t_1, \ldots, t_n) \in T$

Definition. Die Menge der Formeln FO (bezüglich einer Basis BA) ist die kleinste Menge, für die gilt:

(1) $t_1, \ldots, t_n \in T, p \in P^n \Longrightarrow p(t_1, \ldots, t_2) \in FO$

(2) $A, B \in FO \Longrightarrow \neg A, A \vee B \in FO$ (Negation, Disjunktion)

(3) $A \in FO, x \in V \Longrightarrow \exists x\ A \in FO$ ($\exists$ Existenzquantor)

Als Abkürzungen werden noch folgende Zeichen benutzt: $\wedge, \rightarrow, \leftrightarrow, \forall$ (Allquantor) und zwar $\wedge, \rightarrow, \leftrightarrow$ wie bei der Aussagenlogik und

$$\forall x\ A \text{ für } \neg(\exists x\ (\neg A))$$

Beispiel. $BA = (F, P)$
mit $F = F^0 \cup F^1 \cup F^2, P = P^1 \cup P^2$
mit $F^0 = \{a\}, F^1 = \{f\}, F^2 = \{g\}, P^1 = \{q\}, P^2 = \{p\}$

Terme: $x, y, z \ldots \in V$ (Variable)
a (nullstelliges Funktionssymbol)
$g(a, x), f(g(a, x)), g(f(x), g(y, a))$

atomare Formeln: $q(a), q(f(g(a, x))), p(x, y), \ldots$

Formeln: $\forall x\ p(x, f(x)) \wedge q(g(a, z)), \ldots$

Semantik: Um Terme und Formeln zu interpretieren, werden Funktionssymbole als Funktionen und Prädikatensymbole als Prädikate auf einem gewissen Grundbereich interpretiert:

Definition. $\Sigma = (I, w)$ heißt *Struktur* (zur Basis BA) wenn gilt:

(1) I ist eine nichtleere Menge (Individuenbereich)

(2) Für jedes $g \in F^n$ ist $w_g : I^n \rightarrow I$ eine Funktion

(3) Für jedes $p \in P^n$ ist $w_p : I^n \rightarrow \{0, 1\}$ eine Funktion

Beispiel. Für BA wie oben sei $I = \mathbb{N}$ (nat. Zahlen)
$w_a = 2, w_f(n) = n + 1$ (Nachfolgerfunktion)
$w_g(n, m) = n + m$ (Summenfunktion)
w_q Primzahlprädikat, d.h. $w_q(n) = 1$ g.d.w. n prim
$w_p(m, n) = 1$ g.d.w. $m < n$

Definition. Eine Abbildung $V \to I$ heißt *Zustand* oder Belegung der Variablen (mit Individuen).

Zustandsänderung (*updating*)
Sei $\varphi : V \to I$ ein Zustand, $e \in I$
Der Zustand $\varphi[x|e], x \in V$ ist definiert als

$$\varphi[x|e](y) = \texttt{if } y = x \texttt{ then } e \texttt{ else } \varphi(y)$$

(d.h. $\varphi[x|e]$ stimmt mit φ auf allen Variablen $\neq x$ überein und belegt x mit e)
Eigenschaften (offensichtlich):

(1) $\varphi[x|\varphi(x)] = \varphi$

(2) $(\varphi[x|e])[x|d] = \varphi[x|d]$ für alle $e, d \in I$

(3) $x \neq y \Longrightarrow (\varphi[x|e])[y|d] = (\varphi[x|d])[x|e]$ für alle $e, d \in I$

Interpretation von Termen und Formeln.

Definition. Die Interpretation von Termen wird durch eine Abbildung

$$\mathrm{WERT}^T_\Sigma : T \times (V \to I) \to I$$

definiert durch:

(1) $\mathrm{WERT}^T_\Sigma(x, \varphi) = \varphi(x)$ für $x \in V$

(2) $\mathrm{WERT}^T_\Sigma(f(t_1, \ldots, t_n), \varphi) = w_f(\mathrm{WERT}^T_\Sigma(t_1, \varphi), \ldots, \mathrm{WERT}^T_\Sigma(t_n, \varphi))$
$t_1, \ldots, t_n \in T, f \in F^n$

Definition. Die Interpretation von Formeln wird durch eine Abbildung:

$$\mathrm{WERT}^{FO}_\Sigma : FO \times (V \to I) \to \{1, 0\}$$

definiert durch:

(1)
$$\mathrm{WERT}^{FO}_\Sigma(p(t_1, \ldots, , t_n), \varphi) = w_p(\mathrm{WERT}^T_\Sigma(t_1, \varphi), \ldots, \mathrm{WERT}^T_\Sigma(t_n, \varphi))$$
für alle atomaren Formeln $p(t_1, \ldots, t_n) \in FO$

(2)
$$\mathrm{WERT}^{FO}_\Sigma(\neg A, \varphi) = 1 \Longleftrightarrow \mathrm{WERT}^{FO}_\Sigma(A, \varphi) = 0$$
$$\mathrm{WERT}^{FO}_\Sigma(A \vee B, \varphi) = 1 \Longleftrightarrow \mathrm{WERT}^{FO}_\Sigma(A, \varphi) = 1 \text{ oder } \mathrm{WERT}^{FO}_\Sigma(B, \varphi) = 1$$

(3)
$$\mathrm{WERT}^{FO}_\Sigma(\exists x\ A, \varphi) = 1 \Longleftrightarrow \text{ Es gibt ein } e \in I, \text{ so dass } \mathrm{WERT}^{FO}_\Sigma(A, \varphi[x|e]) = 1$$

Bemerkung. Für die Abkürzungen $\wedge, \rightarrow, \leftrightarrow, \forall$ gilt dann (Beweis einfach)

(4)
$$\mathrm{WERT}_\Sigma^{FO}(A \wedge B, \varphi) = 1 \Longleftrightarrow \mathrm{WERT}_\Sigma^{FO}(A, \varphi) = 1 \textbf{ und } \mathrm{WERT}_\Sigma^{FO}(B, \varphi) = 1$$

(5)
$$\mathrm{WERT}_\Sigma^{FO}(A \rightarrow B, \varphi) = 1 \Longleftrightarrow \mathrm{WERT}_\Sigma^{FO}(A, \varphi) = 0 \text{ oder } \mathrm{WERT}_\Sigma^{FO}(B, \varphi) = 1$$

(6)
$$\mathrm{WERT}_\Sigma^{FO}(A \leftrightarrow B, \varphi) = 1 \Longleftrightarrow \mathrm{WERT}_\Sigma^{FO}(A, \varphi) = \mathrm{WERT}_\Sigma^{FO}(B, \varphi)$$

(7)
$$\mathrm{WERT}_\Sigma^{FO}(\forall x\ A, \varphi) = 1 \Longleftrightarrow \text{ Für alle (jedes) } e \in I \text{ ist } \mathrm{WERT}_\Sigma^{FO}(A, \varphi[x|e]) = 1$$

Beispiele: Mit obiger Basis und Struktur Σ:

$$\begin{aligned}
\text{Term } a &: \mathrm{WERT}_\Sigma^{FO}(a, \varphi) = w_a = 2 \\
\text{Term } f(a) &: \mathrm{WERT}_\Sigma^{FO}(f(a), \varphi) = w_f(\mathrm{WERT}_\Sigma^{FO}(a, \varphi)) = w_f(2) = 3 \\
\text{Term } g(a,x) &: \mathrm{WERT}_\Sigma^{FO}(g(a,x), \varphi) = w_g(\mathrm{WERT}_\Sigma^{FO}(a, \varphi), WERT_\Sigma^{FO}(x, \varphi)) = 2 + \varphi(x)
\end{aligned}$$

Formel: $\forall x\ \exists y\ p(x,y) =: \alpha$

$$\begin{aligned}
\underline{\mathrm{WERT}_\Sigma^{FO}(\alpha, \varphi) = 1} &= \text{Für jedes } m \in \mathbb{N} \text{ ist } \mathrm{WERT}_\Sigma^{FO}(\exists y\ p(x,y), \varphi[x|m]) = 1 \\
&= \text{Für jedes } m \in \mathbb{N} \text{ gibt es ein } n \in \mathbb{N} \text{ mit } \mathrm{WERT}_\Sigma^{FO}(p(x,y), \varphi[y|n]) = 1 \\
&= \text{Für jedes } m \in \mathbb{N} \text{ gibt es ein } n \in \mathbb{N} \text{ mit } w_p(m,n) = 1 \\
&= \text{Für jedes } m \in \mathbb{N} \text{ gibt es ein } n \in \mathbb{N} \text{ mit } m < n
\end{aligned}$$

D.h. α hat den Wert 1 unabhängig von φ (für jeden Zustand φ).
Dagegen die Formel $\forall x\ p(x, f(x)) \wedge q(g(a,z))$.

Hier ist $\mathrm{WERT}_\Sigma^{FO}\ (\forall x\ p(x, f(x)), \varphi) = 1$ für jedes φ - aber

$$\begin{aligned}
\mathrm{WERT}_\Sigma^{FO}\ (q(g(a,z)), \varphi) &= w_q(\mathrm{WERT}_\Sigma^{FO}\ (g(a,z), \varphi)) \\
&= w_q(w_g(\mathrm{WERT}_\Sigma^{FO}(a, \varphi), \mathrm{WERT}_\Sigma^{FO}(z, \varphi))) \\
&= w_q(2 + \varphi(z))
\end{aligned}$$

Z.B. bei $\varphi(z) = 0$ oder $\varphi(z) = 3$ ist $\mathrm{WERT}_\Sigma^{FO} = 1$, bei $\varphi = 2$ ist $\mathrm{WERT}_\Sigma^{FO} = 0$.

Definition.

- $A \in FO$ heißt *erfüllbar in der Struktur* $\Sigma = (I, w)$, wenn es einen Zustand $\varphi : V \rightarrow I$ gibt, so dass $\mathrm{WERT}_\Sigma^{FO}(A, \varphi) = 1$ gilt ($A \in FO_{ef-\Sigma}$)
- $A \in FO$ heißt *erfüllbar*, wenn es eine Struktur Σ gibt, in der A erfüllbar ist ($A \in FO_{ef}$)
- $A \in FO$ heißt *gültig in* Σ (A gilt in Σ, Σ ist Modell für A) wenn für alle Zustände $\varphi : V \rightarrow I$ $\mathrm{WERT}_\Sigma^{FO}\ (A, \varphi) = 1$ ($A \in FO_{ag-\Sigma}$)
- $A \in FO$ heißt (allgemein) gültig, wenn sie in allen (zu A passenden) Strukturen gültig ist ($A \in FO_{ag}$) (Allgemein gültige Formeln heißen auch *Tautologien*)

Beispiel. $\forall x\ p(x, f(x))$ ist gültig in Σ, aber nicht allgemein gültig.
$q(g(a,z))$ ist erfüllbar in Σ, also erfüllbar.

Offenbar gilt:

(1) $A \in FO_{ef-\Sigma} \iff \neg A \notin FO_{ag-\Sigma}$

(2) $A \in FO_{ag-\Sigma} \iff \neg A \notin FO_{ef-\Sigma}$

(3) $A \in FO_{ef} \iff \neg A \notin FO_{ag}$

(4) $A \in FO_{ag} \iff \neg A \notin FO_{ef}$

Definition. $A, B \in FO$ heißen (logisch) äquivalent, wenn $A \leftrightarrow B \in FO_{ag} (A \equiv B)$
$A, B \in FO$ heißen erfüllbarkeitsgleich, wenn

$$A \in FO_{ef} \iff B \in FO_{ef}$$

Bemerkung. $A \equiv B$ g.d.w. Für jedes Σ und jedes φ gilt

$$\mathrm{WERT}_\Sigma(A, \varphi) = \mathrm{WERT}_\Sigma(B, \varphi)$$

Gesetze der Prädikatenlogik

- Gesetze für Formeln mit aussagenlogischem Aufbau: Das sind Formeln, die aus Teilformeln mittels $\vee, \neg$ zusammengesetzt sind, z.B. $(\forall x\ A) \vee (\exists x\ \neg B)$.
  Viele Gesetzte der Logik gelten für Formeln mit aussagenlogischem Aufbau, d.h. sie „*betreffen Qualifizierungen nicht*".
  Beispiele sind alle in der Aussagenlogik angegebenen Gesetze.

- *Gesetze für Quantoren:*

In der Formel $\exists x\ r(x, y)$ bezieht sich der Quantor $\exists$ auf x (x ist an diesen Quantor gebunden), während y an keinen Quantor gebunden ist; y ist *frei*.
Man definiert die Menge $fr(A)$ bzw. $fr(t)$ der freien Variablen eines Termes t bzw. einer Formel A *induktiv*.

(1) $fr(x) = \{x\}\ ,\ x \in V$

(2)
$$fr(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} fr(t_i)$$

(3)
$$fr(p(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} fr(t_i)$$

(4) $fr(\neg A) = fr(A)$

(5) $fr(A \vee B) = fr(A) \cup fr(B)$

(6) $fr(\exists x\ A) = fr(A) \setminus \{x\}$

$x \in fr(A)$ heißt frei in A.
$x \in V$ heißt gebunden in A, wenn es eine Teilformel von A der Form $\exists x\ B$ (oder $\forall x\ B$) gibt.

Bemerkung. $x \in V$ kann in einer Formel zugleich frei und gebunden vorkommen, z.B. $p(x) \vee \exists x\ B$. hier ist $x \in fr(p(x) \vee \exists x\ B)$ und x auch gebunden in der Formel.
Man kann das vermeiden, indem man in der Teilformel $\exists x\ B$ mittels „*Umbenennung*“ x durch eine neue Variable y ersetzt, genauer wird in B jedes freie Vorkommen von x durch y ersetzt; das Ergebnis sei $B[x|y]$. Es gilt dann

$$\exists x\ B \equiv \exists y\ B[x|y] \qquad \text{(Beweis mit Induktion)}$$

Man nennt das *gebundene Umbenennung.*

Erneut $\exists x\ r(x,y)$. Man hat

$$\text{WERT}(\exists x\ r(x,y), \varphi) = 1$$

g.d.w. Es gibt $e \in I$ mit $\text{WERT}(r(x,y), \varphi[x|e]) = 1$
g.d.w. Es gibt $e \in I$ mit $w_r(e, \varphi(y)) = 1$
d.h. nur der Wert von y (bei φ) (y ist frei!) beeinflusst den Wert der Formel.
Es gilt allgemein:

Satz 3.5. Wenn $x \notin fr(A)$ (A ist Formel oder Term), so gilt für jedes $e \in I$ und für jedes $\varphi : V \to I$ $\text{WERT}(A, \varphi) = \text{WERT}(A, \varphi[x|e])$ (d.h. dann hängt der Wert nicht von $\varphi(x)$ ab!) (o.B.)

Folgerung. Wenn $fr(A) = \varnothing$, so ist der Wert von A unabhängig von φ

Definition. Eine Formel mit $fr(A) = \varnothing$ heißt *Aussage* oder geschlossene Formel.

Folgerung. Wenn A eine Aussage ist, so gilt

$$A \in FO_{ag-\Sigma} \Longleftrightarrow A \in FO_{ef-\Sigma}$$

Es gelten folgende Gesetze „*über Quantoren*“:

(1)

$$\begin{aligned} \forall x\ \forall y\ A &\equiv \forall y\ \forall x\ A \\ \exists x\ \exists y\ A &\equiv \exists y\ \exists x\ A \\ \exists x\ \forall y\ A &\to \forall \exists x\ A \in FO_{ag} \end{aligned}$$

(Umkehrung gilt nicht: Jedes Kind hat eine Mutter, aber es gibt keine Frau, die Mutter aller Kinder ist.)

(2) Falls $x \notin fr(A)$

$$\begin{aligned} A &\equiv \exists x\ A \\ A &\equiv \forall x\ A \end{aligned}$$

Beweis von (2), erste Zeile

$$\text{WERT}(\exists x\ A, \varphi) = 1$$

g.d.w. Es gibt $e \in I$ mit $\text{WERT}(A, \varphi[x|e]) = 1$
g.d.w. Es gibt $e \in I$ mit $\text{WERT}(A, \varphi) = 1$
Gilt für beliebige φ, also Behauptung.

(3)

$$\begin{aligned} \neg\forall x\ A &\equiv \exists x\ \neg A \\ \neg\exists x\ A &\equiv \forall x\ \neg A \end{aligned}$$

(4) Falls x in B nicht frei vorkommt

$$\begin{aligned}(\forall x\ A) \vee B &\equiv \forall x\ (A \vee B)\\ (\exists x\ A) \wedge B &\equiv \exists x\ (A \wedge B)\end{aligned}$$

(5)

$$\begin{aligned}\forall x\ A \wedge \forall x\ B &\equiv \forall x\ (A \wedge B)\\ \exists x\ A \vee \exists x\ B &\equiv \exists x\ (A \vee B)\end{aligned}$$

Es gilt **nicht:**

$$\begin{aligned}\forall x\ A \vee \forall x\ B &\equiv \forall x\ (A \vee B)\\ \exists x\ A \wedge \exists x\ B &\equiv \exists x\ (A \wedge B)\end{aligned}$$

3.2.1 Normalformen

Pränexe Normalformen (PNF)

Definition. Die Formel A' ist in PNF, wenn gilt

(1) $A' = Q_1x_1 \ldots Q_nx_nM$ mit $n \geq 0$, $Q_i \in \{\exists, \forall\}$

(2) $x_1, \ldots, x_n \in fr(M)$

(3) M ist quantorfrei

M heißt Matrix und $Q_1x_1 \ldots Q_nx_n$ Präfix von A'

Satz 3.6. Es gibt einen Algorithmus, mit dem man zu beliebigem $A \in FO$ ein $A' \in FO$ in PNF konstruieren kann mit $A \equiv A$. A' heißt dann eine PNF von A.

Beweis. Es werden sukzessive Umformungen durchgeführt, die wegen obiger Gesetze und des (auch hier gültigen) Ersetzbarkeitstheorems zu äquivalenten Formeln führen.

Schritt 1: Ersetze in A alle Teilformen der Form $\forall x\ B$ oder $\exists x\ B$ mit $x \notin fr(B)$ durch B.

Schritt 2: Ersetze die Abkürzungen $\rightarrow$ und $\leftrightarrow$ entsprechend ihren Definitionen durch $\neg, \wedge, \vee$

Schritt 3: Ziehe $\neg$ nach Innen durch Ersetzen von:

$$\begin{aligned}\neg\exists x\ B &\quad \text{durch} \quad \forall x\ \neg B\\ \neg\forall x\ B &\quad \text{durch} \quad \exists x\ \neg B\\ \neg\neg B &\quad \text{durch} \quad B\\ \neg(B \vee C) &\quad \text{durch} \quad (\neg B) \wedge (\neg C)\\ \neg(B \wedge C) &\quad \text{durch} \quad (\neg B) \vee (\neg C)\end{aligned}$$

bis das nicht mehr möglich ist.

Schritt 4: (Gebundene Umbenennungen)
Benenne Variablen so um, dass verschiedene Quantoren verschiedene Variablen besitzen, die alle von den in A frei vorkommenden Variablen verschieden sind.

Schritt 5: Ziehe Quantoren heraus.

□

Beispiel.

$$
\begin{aligned}
A &= \forall y\ (\forall x\ \forall y\ p(x,y) \to \exists x\ r(x,y)) \\
A_1 &= \forall y\ (\neg\forall x\ \forall y\ p(x,y) \vee \exists x\ r(x,y)) && \text{Schritt 2} \\
A_2 &= \forall y\ (\exists x\ \exists y\ p(x,y) \vee \exists x\ r(x,y)) && \text{Schritt 3} \\
A_3 &= \forall y\ (\exists x\ \exists z\ p(x,z) \vee \exists u\ r(u,y)) && \text{Schritt 4} \\
A_4 &= \underbrace{\forall y\ \exists x\ \exists z\ \exists u}_{\text{Präfix}}\ \underbrace{(\neg p(x,z) \vee r(u,y))}_{\text{Matrix}} && \text{Schritt 5}
\end{aligned}
$$

Bemerkung. PNF ist nicht eindeutig bestimmt

Universelle Normalform (UNF)
Man möchte alle Existenzquantoren beseitigen. Das geht nicht, wenn die Normalform äquivalent (zur ursprünglichen Formel) sein soll.
Deshalb fordert man nur die Erfüllbarkeitsgleichheit.
Der Algorithmus beruht auf folgender Tatsache:

$$\forall x_1 \ldots \forall x_n\ \exists y\ A \in FO_{ef} \text{ g.d.w. } \forall x_1 \ldots \forall x_n\ Ay[h(x_1,\ldots,x_n)] \in FO_{ef}$$

wobei $Ay[h(x_1,\ldots,x_n)]$ aus A entsteht, indem y in A durch $h(x_1,\ldots,x_n)$ ersetzt wird mit einem **neuen** n-stelligen Funktionssymbol h.
Dies heißt *Skolemisierung* (Skolem) (o.B.)

Definition. Die Formel A' ist in UNF, wenn gilt

(1) A' ist in PNF

(2) A' ist eine Aussage

(3) im Präfix von A' gibt es keine $\exists$-Quantoren.

Satz 3.7. Es gibt einen Algorithmus, mit dem man zu beliebigem $A \in FO$ ein A' in UNF konstruieren kann mit A ist erfüllbar $\iff$ A' ist erfüllbar.
A' heißt eine UNF von A.

Beweis.

Schritt 0: Falls $fr(A) = \{x_1,\ldots,x_k\}$, so ersetze A durch $\exists x_1,\ldots,\exists x_k\ A$.

Schritt 1 bis 4: wie bei PNF

Schritt 5: Eliminiere $\exists$-Quantoren entsprechend $(*)$

Schritt 6: Ziehe $\forall$-Quantoren heraus

□

Beispiel.

$$\forall x\ \forall y\ \ ((p(x) \to \neg r(x,y)) \to \neg\forall x\ \exists z\ (q(x,z) \wedge s(z)))$$

$$
\begin{aligned}
A' &= \forall x\ \forall y\ \ (\neg(\neg p(x) \vee \neg r(x,y)) \vee \neg\forall x\ \exists z\ (q(x,z) \wedge s(z))) \\
A'' &= \forall x\ \forall y\ \ ((p(x) \wedge r(x,y)) \vee \exists x\ \forall z\ (\neg q(x,z) \vee \neg s(z))) \\
A''' &= \forall x\ \forall y\ \ ((p(x) \wedge r(x,y)) \vee \exists u\ \forall z\ (\neg q(u,z) \vee \neg s(z))) \\
A'''' &= \forall x\ \forall y\ \ ((p(x) \wedge r(x,y)) \vee \forall z\ (\neg q(g(x,y),z) \vee \neg s(z))) \ ;\ \text{genaues Funktionssymbol} \\
A''''' &= \forall x\ \forall y\ \forall z\ \ ((p(x) \wedge r(x,y)) \vee \neg q(g(x,y),z) \vee \neg s(z))
\end{aligned}
$$

ist in UNF

3.2.2 Folgern und Ableiten

Semantischer Folgerungsbegriff
Wir wollen wieder die Frage betrachten:
Wann folgt eine Aussage aus einer Menge von Voraussetzungen?

Definition. Sei X eine Menge von Aussagen und A eine Aussage. A heißt Folgerung von X, in Zeichen: $X \vDash A$, wenn für jedes Σ mit $x \subseteq FO_{ag-\Sigma}$ auch $A \in F = FO_{ag-\Sigma}$.

Insbesondere möchte man die Frage $X \vDash A$ algorithmisch beantworten können (aufgrund der syntaktischen Struktur von X und A).

Es gilt aber:

Satz 3.8. Das Problem $X \vDash A$ ist nicht entscheidbar. (o.B.)

aber

Satz 3.9. Das Problem $X \vDash A$ ist semientscheidbar, d.h. es gibt ein Verfahren (TURING-Maschiene), das bei Input von X und A hält mit der Antwort „*Ja*“, wenn $X \vDash A$ gilt.
(Aber: Wenn $\nvDash A$, braucht Verfahren nicht zu halten)

- *Zurückführung auf Nichterfüllbarkeit:*

Satz 3.10. Für Aussagen $A_1, \ldots, A_n, B$ gilt

$$\{A_1, \ldots, A_n\} \vDash B \text{ g.d.w. } A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge \neg B \notin FO_{ef}$$

- *Nach Satz über UNF:* ($U(A)$ sei UNF von A

$$\{A_1, \ldots, A_n\} \vDash B \text{ g.d.w. } U(A_1 \wedge \cdots \wedge A_n \wedge \neg B) \notin FO_{ef}$$

- Für $A \in FO$ mit A in UNF mit Matrix in KNF sei $\mathcal{F}(A)$ die zugehörige Klauselmenge

Beispiel.

$$\forall x\ \forall y\ \forall z\ \underbrace{((p(x,y) \wedge r(x,y)) \vee (\neg q(g(x,y),z) \vee \neg s(z)))}_{\text{Matrix}}$$

$$\equiv A = \forall x\ \forall y\ \forall z\ \underbrace{([p(x,y) \vee q(g(x,y),z) \vee \neg s(z)] \wedge [r(x,y) \vee \neg q(g(x,y),z) \vee \neg s(z)])}_{\text{Matrix in KNF}}$$

$$\mathcal{F}(A) = \{\{p(x,y), \neg q(g(x,y),z), \neg s(z)\}, \{r(xy), \neg q(g(x,y),z), \neg s(z)\}\}$$

Jetzt:

Prädikatenlogische Resolution
Ein *Ausdruck* A sei ein Term oder eine atomare Formel, $V(A)$ sei die Menge der in A vorkommenden Variablen.

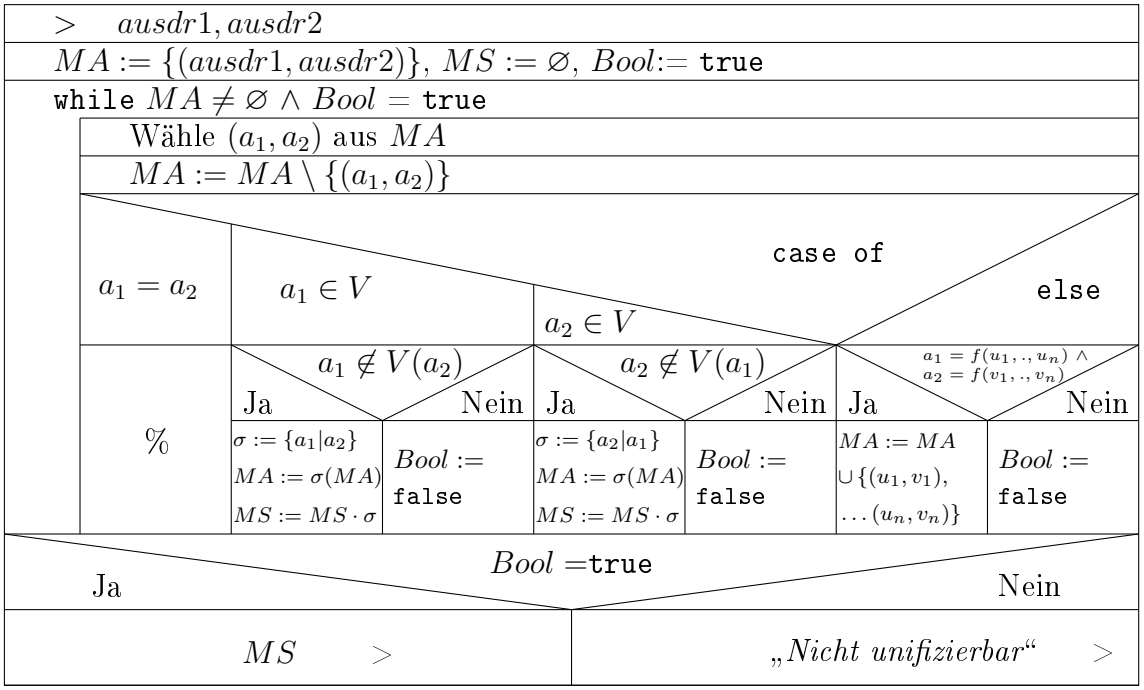
Definition. Seien A_1 und A_2 Ausdrücke. Ein Unifikator von A_1 und A_2 ist eine Substitution σ mit $\sigma(A_1) = \sigma(A_2)$ (eine Substitution $\sigma : V \to T$ ist eine Abbildung, so dass für fast alle $x \in V$ $\sigma(x) = x$ gilt).
Ein allgemeiner Unifikator (a.U) ist ein Unifikator σ von A_1 und A_2 mit der Eigenschaft:
Wenn α ein Unifikator von A_1 und A_2 ist, so gibt es eine Substitution β mit $\alpha = \sigma \cdot \beta$
(d.h. für jeden Ausdruck A git $\alpha(A) = \beta(\sigma(A))$).

Beispiel. Sei $A_1 = p(x, g(y))$, $A_2 = p(x, g(b))$
$\alpha = \{x|a, y|b\}$ ist ein Unifikator von A_1 und A_2.
(„α unifiziert A_1 und A_2“)

$$\sigma = \{y|b\} \text{ ist ein a.U. und } \alpha = \sigma \cdot \{x|a\}$$

Satz 3.11. Es gibt einen Algorithmus, mit dem man für zwei beliebige Ausdrücke A_1 und A_2 feststellen kann, ob sie unifizierbar sind und der im Falle, dass ja, einen a.U. liefert.

Beweis. Der Beweis wird durch Angabe eines Algorithmus geliefert:
Man kann zeigen (und sieht an Beispielen leicht ein), dass der Algorithmus immer terminiert und immer einen a.U. liefert, falls die Ausdrücke unifizierbar sind. □



Beispiel.

a) $A_1 = p(x)$, $A_2 = p(g(x))$

MA	MS	*Bool*	σ	(a_1, a_2)
$\{(p(x), p(g(x)))\}$	$\varnothing$	`true`		$(p(x), p(g(x)))$
$\varnothing$				
$\{(x, g(x))\}$		`false`		$(x, g(x))$
$\varnothing$				$\Rightarrow$ „*Nicht unifizierbar*“

b) $A_1 = p(x, g(x), y)$, $A_2 = p(u, v, h(v))$ mit $x, y, u, v \in V$

MA	MS	*Bool*	σ	(a_1, a_2)
$\{(p(x, g(x), y), p(u, v, h(v))\}$	$\varnothing$	`true`		$(p(x, g(x), y), p(u, v, h(v))$
$\varnothing$				
$\{(x\|u), (g(x), v), (y, h(v))\}$			$\{x\|u\}$	$(x\|u)$
$\{(g(x), v), (y, h(v))\}$				
$\{g(u), v), (y, h(v))\}$	$\{x\|u\}$			$(g(u), v)$
$\{(y, h(v))\}$	$\{x\|u, v\|g(u)\}$		$\{v\|g(u)\}$	
$\{(y, h(g(u)))\}$				$(y, h(g(u))$
$\varnothing$	$\{x\|u, v\|g(u), y\|h(g(u))\}$		$\{y\|h(g(u))\}$	

ist a.U.

Seien jetzt C unnd D Klauseln, σ_C und σ_D „*Umbenennungssubstitutionen*“ mit

$$V(\sigma_C(C)) \cap V(\sigma_D(D)) = \varnothing$$

$$\sigma(C) = \{L, L_1, \ldots, L_n\} \ , \ \sigma_D(D) = \{M, M_1, \ldots, M_m\}$$

Sind Atom (L) und Atom (M) unifizierbar mit a.U. θ und sind $\theta(L)$ und $\theta(M)$ komplementär, so heißt $\{\theta(L_1), \ldots, \theta(L_n), \theta(M_1), \ldots, \theta(M_m)\}$ (binäre) *Resolvente* von C und D.

Beispiel. $C = \{p(x), q(x)\}$, $D = \{\neg p(g(x)), q(a)\}$
Umbenennung: $\sigma_C(C) = \{p(z), q(z)\}$, $\sigma_D = id$.
Atom $(p(z)) = p(z)$, Atom $(\neg p(g(x))) = p(g(x))$
Diese sind unifizierbar mit a.U. $\theta = \{z|g(x)\}$
$\Rightarrow \{q(g(x)), q(a)\}$ ist eine binäre Resolvente von C und D.

Sei $C = \{L, M, L_1, \ldots, L_k\}$ eine Klausel.
Sind L und M unifizierbar mit a.U. θ (d.h. L und M beide Atome oder beide negierte Atome und dann sind die zugehörigen Atome unifizierbar mit a.U. θ), so heißt $\{\theta(L), \theta(L_1), \ldots, \theta(L_k)\}$ ein Faktor von C.

Beispiel. $\{p(x), p(y), r(x, y)\}$ liefert den Faktor $\{p(y), p(y, y)\}$.

Definition. Ist $\mathcal{F}$ eine Klauselmenge, so ist $\mathrm{Res}(\mathcal{F})$ definiert als

$$\begin{aligned} \mathrm{Res}^1(\mathcal{F}) = \mathrm{Res}(\mathcal{F}) &= \mathcal{F} \cup \{R : \ R \text{ ist binäre Resolvente zweier Klauseln aus } \mathcal{F}\} \\ &\quad \cup \{S : \ S \text{ ist Faktor einer Klausel aus } \mathcal{F}\} \\ \mathrm{Res}^{n+1}(\mathcal{F}) &= \mathrm{Res}(\mathrm{Res}^n(\mathcal{F})) \\ \mathrm{Res}^*(\mathcal{F}) &= \bigcup_{n \geq 1} (\mathrm{Res}^n(\mathcal{F})) \end{aligned}$$

Satz 3.12 (Resolutionssatz der Prädikatenlogik)**.** Sei A eine Aussage in UNF, $\mathcal{F}(\mathcal{A})$ ihre Klauselmenge. A ist nicht erfüllbar g.d.w. $\square \in \mathrm{Res}^*(\mathcal{F}(\mathcal{A}))$

Folgerung. Seien $A_1, \ldots, A_n, B$ Aussagen. $\mathcal{F}$ die zur UNF von $A_1 \wedge \cdots \wedge A_n \wedge B$ gehörende Klauselmenge.
Es gilt $A_1, \ldots, A \vDash B$ g.d.w. $\square \in \mathrm{Res}^*(\mathcal{F})$.

Semientscheidungsverfahren:

$$\mathcal{F} \subseteq \mathrm{Res}(\mathcal{F}) \subseteq \mathrm{Res}^2(\mathcal{F}) \subseteq \dots$$

Zum „*Beweis*“:

$\Longleftarrow$: Korrektheit des Kalküls

$\Longrightarrow$: Vollständigkeit des Kalküls

Bemerkung. Offenbar kann man $\mathcal{F}$ bilden, indem man zu $A_1, \dots, A_n, \neg B$ einzeln die UNF's und die Klauseln bildet.

Beispiel.

a) SOKRATES

$$\forall x\ (\underbrace{m(x) \to t(x)}_{A_1}),\ \underbrace{m(s)}_{A_2} \vDash \underbrace{t(s)}_{B}?$$

$$\begin{aligned} A_1 &:\{\neg m(x),\ t(x)\} \\ &\qquad\qquad\quad \searrow \\ A_2 &:\{m(s)\} \qquad \to\ \{t(s)\} \\ &\qquad\qquad\qquad\quad \searrow \\ \neg B &:\{\neg t(s)\} \qquad\qquad \to \square \end{aligned}$$

b) Gilt $\underbrace{\exists x\ \forall y\ r(x,y)}_{A_1} \vDash \underbrace{\forall y\ \exists x\ r(x,y)}_{B}$?

$$\begin{aligned} A_1 &:\{r(a,y)\} \\ &\qquad\quad \searrow \\ \neg B &:\{\neg r(x,b)\} \to \square \end{aligned}$$

c) Gilt dagegen $\underbrace{\forall y\ \exists x\ r(x,y)}_{A_1} \vDash \underbrace{\exists x\ \forall y\ r(x,y)}_{B}$?

$$A_1 :\{r(g(y),y)\}$$

$$\neg B :\{\neg r(x,h(x))\}$$

Aber $r(g(y),y)$ und $r(x,h(x))$ sind nicht unifizierbar!

$$\Longrightarrow \square \notin \mathrm{Res}^*(\mathcal{F})$$

also gilt die Behauptung nicht.

d) *Gruppentheorie*:
In einem abgeschlossenen assoziativen System $(G, \circ)$, in dem die Gleichungen $x \circ u = v$ und $u \circ y = v$ Lösungen haben, gibt es ein Rechtseinselement.

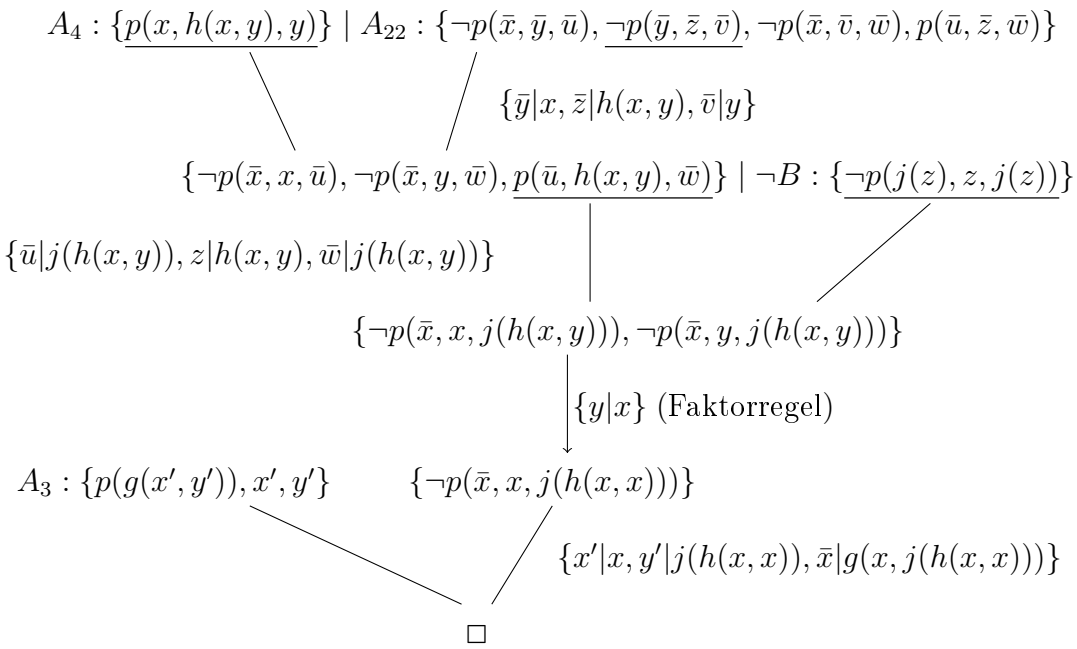
Formal: Prädikatensymbol $p \in P^3 : p(x, y, z)$ wenn $x \circ y = z$
Dann wird

$$
\begin{aligned}
A_1 &: \forall x\ \forall y\ \exists z\ p(x, y, z) && \text{(Abgeschlossenheit)}\\
A_2 &: \forall x\ \forall y\ \forall z\ \forall u\ \forall v\ \forall w\ ((p(x, y, u) \wedge p(y, z, v)) \rightarrow (p(u, z, w) \leftrightarrow p(x, v, w))) \\
&&& \text{(Assoziativität)}\\
A_3 &: \forall x\ \forall y\ \exists z\ p(z, x, y) && \text{(Linkslösung)}\\
A_4 &: \forall x\ \forall y\ \exists z\ p(x, z, y) && \text{(Rechtslösung)}\\
\neg B &: \neg \exists z\ \forall x\ p(x, z, x)
\end{aligned}
$$

$\rightsquigarrow$ Klauseln

$$
\begin{aligned}
A_1 &: \{p(x, y, k(x, y))\}\\
A_{2_1} &: \{\neg p(x, y, u), \neg p(y, z, v), \neg p(u, z, w), p(x, v, w)\}\\
A_{2_2} &: \{\neg p(x, y, u), \neg p(y, z, v), \neg p(x, v, w), p(u, z, w)\}\\
A_3 &: \{p(g(x, y), x, y)\}\\
A_4 &: \{p(x, h(x, y), y)\}\\
\neg B &: \{\neg p(j(z), z, j(z))\}
\end{aligned}
$$

Ein möglicher Resolutionsbeweis geht so:



Natürlich sind für automatische Beweise noch Strategien zu entwickeln, wie Resolventen bzw. Faktoren gebildet werden sollten.

3.2.3 Prolog

Definition. Eine Horn-Klausel ist eine Klausel, wenn mindestens eines ihrer Literale ein Atom ist.
Sie heißt *definit*, wenn genau ein Literal ein Atom ist.

HORN-Klauseln sind die Grundlage von PROLOG. Ein PROLOG-Programm ist eine Menge von definiten Klauseln.
Ist $\{A, \neg B_1, \ldots, \neg B_k\}$ eine definite Klausel, d.h. $A, B_1, \ldots, B_k$ Atome, so kann sie geschrieben werden als

$$B_1 \wedge \cdots \wedge B_k \rightarrow A$$

oder

$$A \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_k$$

und in PROLOG:

$$A :- B_1, \ldots, B_k$$

Das Starten eines PROLOG-Programmes erfolgt durch eine *Anfrage*, d.h. durch die Angabe einer Zielklausel, das ist eine Klausel, die nur aus negierten Atomen besteht:

$$\neg G_1 \vee \cdots \vee G_l \quad , \; G_i \text{ Atome}$$

oder

$$G_1 \wedge \cdots \wedge G_l \rightarrow \square$$

oder in PROLOG:

$$?- G_i, \ldots, G_l$$

Das PROLOG-System testet nun gerade, ob die gegebenen HORN-Klauseln zusammen mit der Zielklausel nicht erfüllbar sind, d.h. also, ob $G_1 \wedge \cdots \wedge G_l$ eine logische Folgerung der HORN-Klauseln (der Datenbasis) ist.

Beispiel. (SOKRATES)

$$\begin{gathered} m(s).; \{m(s)\} \\ t(x) :- m(x); \{t(x), \neg m(x)\} \\ \underbrace{?- t(s).}_{yes}; \{\neg t(s)\} \end{gathered}$$

Über dies gibt PROLOG noch aus, durch welche Substitution für die Variablen der Zielklausel das Ziel erreicht wird (Antwortsubstitution).

Beispiel. Wir betrachten folgende Aussagen:

- Paul ist Wissenschaftler
- Franz ist Wissenschaftler
- Paul ist Deutscher
- Franz ist Franzose
- jeder Wissenschaftler ist Logiker

Anfrage: Gibt es unter den Franzosen Logiker?
formalisiert:

		PROLOG
$wiss(Paul)$	$\{wiss(Paul)\}$	$wiss(Paul).$
$wiss(Franz)$	$\{wiss(Franz)\}$	$wiss(Franz).$
$dt(Paul)$	$\{dt(Paul)\}$	$dt(Paul):$
$frz(Franz)$	$\{frz(Franz)\}$	$frz(Franz).$
$\forall x(wiss(x) \rightarrow log(x))$	$\{\neg wiss(x), log(x)\}$	$log(x) :- wiss(x).$
$\neg\exists x(frz(x) \wedge log(x))$	$\{\neg frz(x), \neg log(x)\}$	$?- frz(x), log(x).$
		$x = Franz$

Literaturverzeichnis

[Sch:TIk] Schöning: Theoretische Informatik kurzgefasst

[Hop:EidA] Hoperoft; Ullman: Einführung in die Automatentheorie

[Weg:TI] Wegener: Theoretische Informatik

[Wag:Ti] Wagener: Theoretische Informatik

[Sch:LfI] Schöning: Logik für Informatiker