Formale Sprachen und Automaten

Robin Rausch, Ozan Akzebe, Florian Maslowski

21. November 2022

Inhaltsverzeichnis

1 Grundlagen

1.1 Alphabet

Ein Alphabet Σ ist eine nicht-leere Menge von Symbolen(Zeichen, Buchstaben). Beispiel: $\Sigma_{ab} = a, b$

1.2 Wort

Ein Wort w über dem Alphabet Σ(Sigma) ist eine endliche Folge von Symbolen aus Σ. Das Wort $w = abaabab$ wurde beispielsweise aus dem Alphabet Σ_{ab} gebildet.
Die Länge eines Wortes kann durch Betragsstriche angegeben werden. Beispiel: $|w| = 7$
Ebenso kann man die Anzahl bestimmter Symbole in einem Wort bestimmen: $|w|_b = 3$
Ein einzelnes Zeichen kann durch eckige Klammern angegeben werden: $w[2] = b$
Wörter können bliebig konkateniert werden(hintereinanderschreiben ohne abstand): $w_1 w_2 = abbabaab$ mit $w_1 = abba$ und $w_2 = baab$.
Wörter dürfen auch potenziert werden: $w^3 = abaabababaababababaabab = www$
Das leere Wort lautet ε.

1.3 Formale Sprachen

Eine formale Sprache L über einem Alphabet Σ ist eine Menge von Wörtern aus $\Sigma^* : L \subseteq \Sigma^*$. Eine Sprache kann sowohl endlich als auch unendlich sein.
Beispiel: $L_1 = \{w \in \Sigma^*_{bin} | \, |w| \geqslant 2 \wedge w[|w| - 1] = 1\}$ ist die Menge aller Binärwörter, an deren vorletzter Stelle 1 steht.
Das Produkt zweier formaler Sprachen: $L_1 \cdot L_2 = \{abac, abcb, bcac, bccb\}$ mit $L_1 = \{ab, bc\}$ und $L_2 = \{ac, cb\}$.
Sprachen können ebenfalls potenziert werden: $L^2 = \{ab, ba\} \cdot \{ab, ba\} = \{abab, abba, baab, baba\}$

1.4 Kleene Stern

Für ein Alphabet Σ und eine formale Sprache $L \subseteq \Sigma^*$ ist der Operator Kleene Stern wie folgt definiert: $L^* = \bigcup\limits_{n \in \mathbb{N}} L^n$.

Beispiel: Sei $L_1 = \{ab, ba\}$, dann $L^* = \{\varepsilon, ab, ba, abab, abba, baab, baba, ababab, ...\}$.

1.5 Überblick

Typ	Sprachklasse	Grammatik	Maschinenmodell
0	rekursiv aufzählbar	unbeschränkt	Turing-Maschine
1	kontextsensitiv	monoton	linear beschränkter Automat
2	kontextfrei	kontextfrei	Kellerautomat
3	regulär	rechtslinear	endlicher Automat

2 Reguläre Sprachen und endliche Ausdrücke

2.1 Reguläre Ausdrücke

Ein regulärer Ausdruck über Σ beschreibt eine formale Sprache.
Die Menge aller regulären Ausdrücke über Σ ist eine formale Sprache.

Beispiel: Sprache aller Wörter über Σ_{abc}, die nur aus genau zwei Symbolen bestehen:
Ausdruck: $r_1 = (a + b + c)(a + b + c)$
Sprache: $\mathcal{L}(r_1) = \{w \in \Sigma^*_{abc} \mid |w| = 2\}$
Operatoren:

$$
\begin{array}{rcll}
r_1 + r_2 & \equiv & r_2 + r_1 & \text{Kommutativität von } + \\
(r_1 + r_2) + r_3 & \equiv & r_1 + (r_2 + r_3) & \text{Assoziativität von } + \\
(r_1 r_2) r_3 & \equiv & r_1 (r_2 r_3) & \text{Assoziativität von } \cdot \\
\emptyset r \equiv r\emptyset & \equiv & \emptyset & \text{Absorbierendes Element für } \cdot \\
\varepsilon r \equiv r\varepsilon & \equiv & r & \text{Neutrales Element für } \cdot \\
\emptyset + r & \equiv & r & \text{Neutrales Element für } + \\
(r_1 + r_2) r_3 & \equiv & r_1 r_3 + r_2 r_3 & \text{Distributivität links} \\
r_1 (r_2 + r_3) & \equiv & r_1 r_2 + r_1 r_3 & \text{Distributivität rechts} \\
r + r & \equiv & r & \text{Idempotenz von } + \\
(r^*)^* & \equiv & r^* & \text{Idempotenz von } ^* \\
\emptyset^* & \equiv & \varepsilon & \\
\varepsilon^* & \equiv & \varepsilon & \\
\varepsilon + r^* r & \equiv & r^* & \\
(\varepsilon + r)^* & \equiv & r^* & \\
r^* r & \equiv & r r^* &
\end{array}
$$

Nicht alle Operatoren sind für alle Typen zulässig:

	Vereinigung	Konkatenation	Potenz	Kleene-Stern
Wörter	✗	$w_1 \cdot w_2$	w^n	✗
Sprachen	$L_1 \cup L_2$	$L_1 \cdot L_2$	L^n	L^*
Reguläre Ausdrücke	$r_1 + r_2$	$r_1 \cdot r_2$	✗	r^*

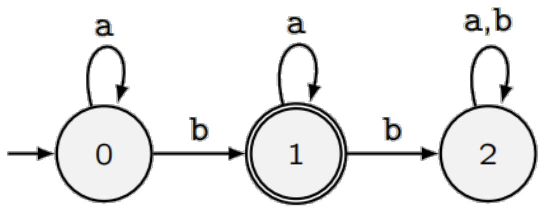
2.2 Endliche Automaten

Endliche Automaten sind eine andere Darstellung einer regulären Sprache. Endliche Ausdrücke lassen sich in Reguläre Ausdrücke umformen. Genauso auch anders herum.
Endliche Automaten erkennen regulären Sprachen. Endliche Ausdrücke lassen sich in Reguläre Ausdrücke umformen. Genauso auch anders herum.
Endliche Automaten lassen sich sowohl deterministisch als auch nicht-deterministisch darstellen.

2.2.1 Deterministische endliche Automaten(DEA)

Ein DEA hat endlich viele Zustände. Jeder mögliche Übergang muss hierbei behandelt werden können. D.h. für das Alphabet Σ_{ab} muss von jedem Zustand sowohl ein a, als auch ein b Übergang gegeben sein. Er terminiert wenn das Wort zu ende ist und dabei ein Endzustand erreicht ist.
Der DEA lässt sich durch folgendes 5-Tupel darstellen:
$\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ mit den Komponenten:
Q ist eine endliche Menge von Zuständen
Σ ist ein endliches Alphabet
$\delta : Q \times \Sigma \rightarrow Q$ ist die Übergangsfunktion
$q_0 \in Q$ ist der Startzustand
$F \subseteq Q$ ist die Menge der Endzustände

Beispiel:



$\mathcal{A}_b = (Q, \Sigma, \delta, q_0, F)$ mit

- $Q = \{0, 1, 2\}$
- $\Sigma = \Sigma_{ab}$
- $\delta(0, a) = 0; \delta(0, b) = 1, \delta(1, a) = 1; \delta(1, b) = \delta(2, a) = \delta(2, b) = 2$
- $q_0 = 0$
- $F = \{1\}$

Zustand 2: „Mülleimerzustand“ (junk state), d.h. kein Wort wird mehr akzeptiert

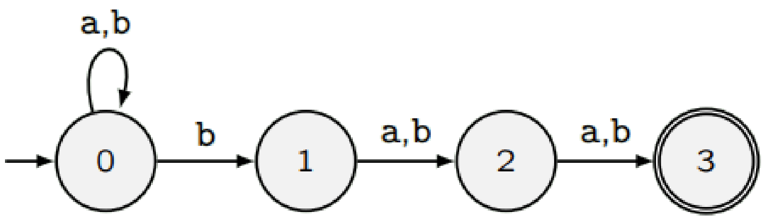
Run/Konfigurationsfolge: 2er Tupel: (q, w) mit $q \in Q \wedge w \in \sum^*$

2.2.2 Nicht-deterministische endliche Automaten(NEA)

Ein NEA hat endlich viele Zustände. Nicht jeder mögliche Übergang muss hierbei behandelt werden. D.h. für das Alphabet Σ_{ab} reicht es, nur den a-Übergang, bzw. nur den b-Übergang zu besitzen(oder keinen). Der Automat beginnt im Startzustand und muss im Endzustand enden. Wenn der Automat sich nicht in einem Endzustand befindet, befindet sich das Wort nicht in der Sprache, welche vom Automaten abgebildet wird. Zudem gibt es ε-Übergänge, diese können jederzeit verwendet werden ohne ein Eingabesymbol.
Der NEA lässt sich durch folgendes 5-Tupel darstellen:
$\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ mit den Komponenten:
Q ist eine endliche Menge von Zuständen
Σ ist ein endliches Alphabet

Δ ist eine Relation über $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$
$q_0 \in Q$ ist der Startzustand
$F \subseteq Q$ ist die Menge der Endzustände

Beispiel:



$\mathcal{A}_n = (Q, \Sigma, \Delta, q_0, F)$ mit

$$\begin{aligned} Q &= \{0, 1, 2, 3\} \\ \Sigma &= \Sigma_{ab} \\ \Delta &= \{(0, \mathbf{a}, 0), (0, \mathbf{b}, 0), (0, \mathbf{b}, 1), \\ &\quad (1, \mathbf{a}, 2), (1, \mathbf{b}, 2), \\ &\quad (2, \mathbf{a}, 3), (2, \mathbf{b}, 3)\} \\ q_0 &= 0 \\ F &= \{3\} \end{aligned}$$

	$\mathcal{A}_n$	a	b	ε
$\rightarrow$	0	$\{0\}$	$\{0, 1\}$	$\{\}$
	1	$\{2\}$	$\{2\}$	$\{\}$
	2	$\{3\}$	$\{3\}$	$\{\}$
$*$	3	$\{\}$	$\{\}$	$\{\}$

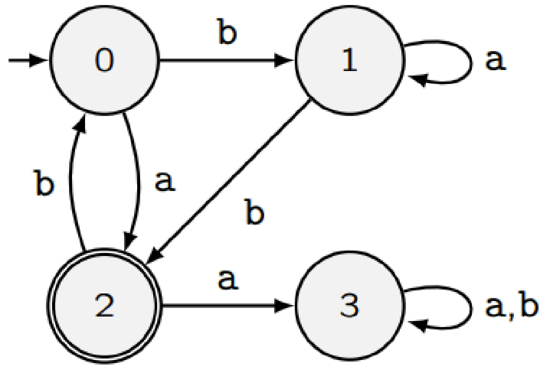
Run/Konfigurationsfolge: 2er Tupel: (q, w) mit $q \in Q \wedge w \in \sum^*$

2.2.3 Komplement eines EAs

Wird durch vertauschen von End- und Nichtendzuständen erzeugt.

2.2.4 Transformation DEA zu RA

Um einen DEA zu einem RA umzuformen müssen zuerst die Gleichungen der jeweiligen Zustände aufgestellt und vereinfacht werden:



- $L_0 \equiv aL_2 + bL_1$
- $L_1 \equiv aL_1 + bL_2$
- $L_2 \equiv aL_3 + bL_0 + \varepsilon$
- $L_3 \equiv (a+b)L_3$

$$
\begin{array}{lcll}
L_3 & \equiv & (a+b)L_3 + \emptyset & \text{[neutrales Element]} \\
 & \equiv & (a+b)^*\emptyset & \text{[Arden]} \\
 & \equiv & \emptyset & \text{[absorbierendes Element]} \\
L_2 & \equiv & a\emptyset + bL_0 + \varepsilon & \text{[einsetzen } L_3\text{]} \\
 & \equiv & \emptyset + bL_0 + \varepsilon & \text{[absorbierendes Element]} \\
 & \equiv & bL_0 + \varepsilon & \text{[neutrales Element]} \\
L_1 & \equiv & aL_1 + b(bL_0 + \varepsilon) & \text{[einsetzen } L_2\text{]} \\
 & \equiv & a^*b(bL_0 + \varepsilon) & \text{[Arden]}
\end{array}
$$

Da L_2 der Endzustand ist, bekommt die Gleichung ε hinzuaddiert!
Folglich kann man die gekürzten Gleichungen in einander einsetzen um bis zum Anfangszustand zu kommen:

- $L_0 \equiv aL_2 + bL_1$
- $L_1 \equiv a^*b(bL_0 + \varepsilon)$
- $L_2 \equiv bL_0 + \varepsilon$
- $L_3 \equiv \emptyset$

Der vereinfachte Anfangszustand L_0 ist dann der RA zum zugehörigen DEA:

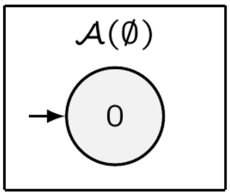
$$
\begin{array}{lcll}
L_0 & \equiv & a(bL_0 + \varepsilon) + b(a^*b(bL_0 + \varepsilon)) & \text{[einsetzen } L_1, L_2\text{]} \\
 & \equiv & abL_0 + a + ba^*bbL_0 + ba^*b & \text{[Distributivgesetz]} \\
 & \equiv & (ab + ba^*bb)L_0 + a + ba^*b & \text{[Kommutativ-,Distributivgesetz]} \\
 & \equiv & (ab + ba^*bb)^*(a + ba^*b) & \text{[Arden]}
\end{array}
$$

$$\text{Ergebnis: } \mathcal{L}(\mathcal{A}) = \mathcal{L}((ab + ba^*bb)^*(a + ba^*b))$$

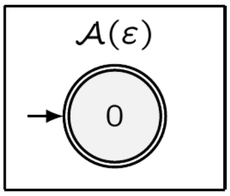
2.2.5 RA zu NEA

Aus den elementaren RA können einfach NEAs erstellt werden.

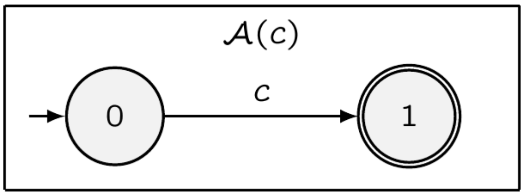
1. $\mathcal{A}(\emptyset) = (\{0\}, \Sigma, \{\}, 0, \{\})$



2. $\mathcal{A}(\varepsilon) = (\{0\}, \Sigma, \{\}, 0, \{0\})$

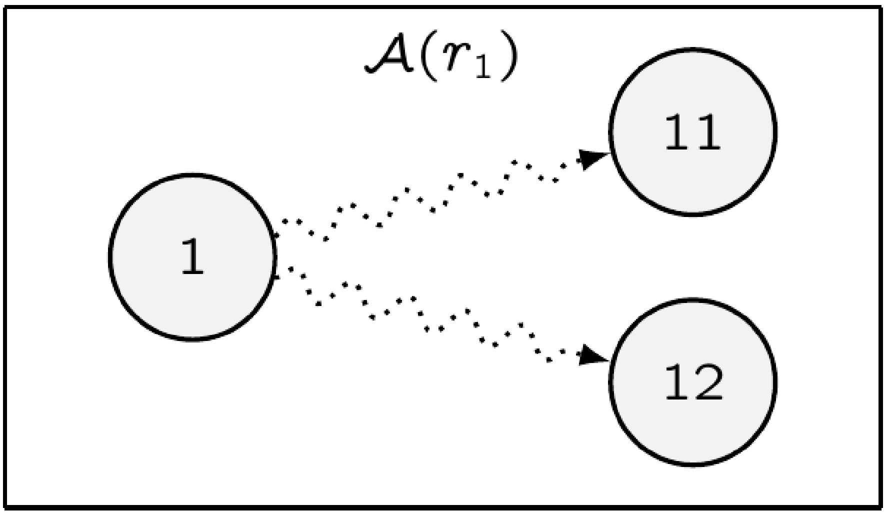


3. $\mathcal{A}(c) = (\{0, 1\}, \Sigma, \{(0, c, 1)\}, 0, \{1\})$ für alle $c \in \Sigma$



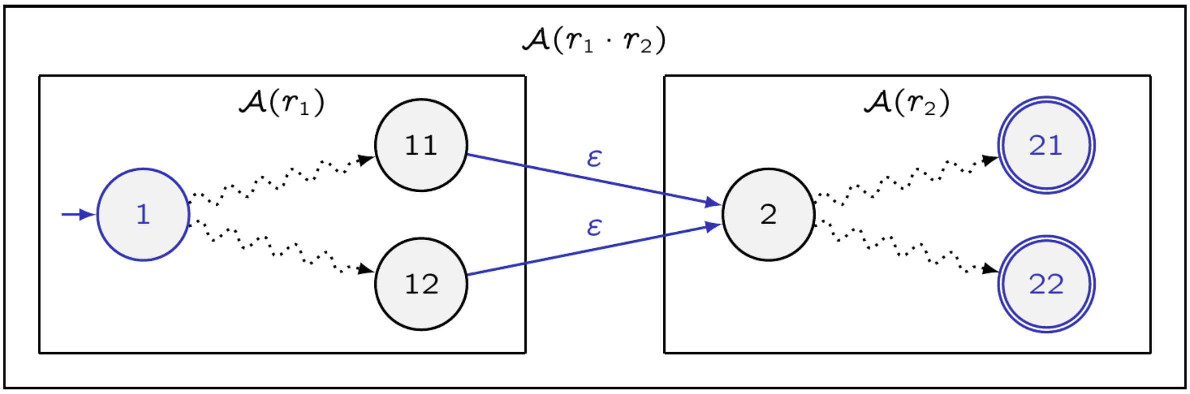
Bei komplexen RAs werden die RAs als "BlackBox"dargestellt, dabei werden die Übergänge gepunktet dargestellt.

$$\mathcal{A}(r_1) = (Q_1, \Sigma, \Delta_1, 1, \{11, 12, \ldots\})$$



- $\mathcal{A}(r_1) = (Q_1, \Sigma, \Delta_1, 1, \{11, 12, \ldots\})$
- $\mathcal{A}(r_2) = (Q_2, \Sigma, \Delta_2, 2, \{21, 22, \ldots\})$

4 $\mathcal{A}(r_1 \cdot r_2) = (Q_1 \cup Q_2, \Sigma, \Delta_1 \cup \Delta_2 \cup \{(11, \varepsilon, 2), (12, \varepsilon, 2)\}, 1, \{21, 22\})$

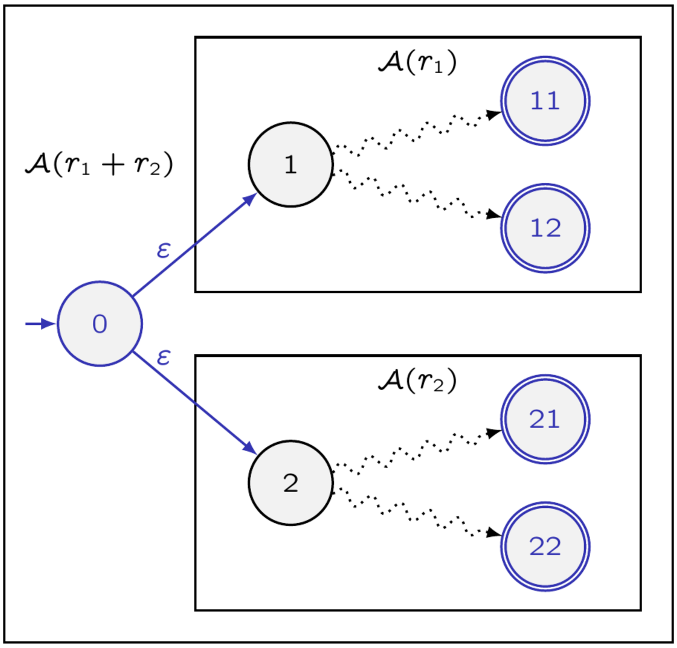


- Startzustand 1
- ε-Übergänge von 11 und 12 zu 2
- Endzustände 21 und 22

- $\mathcal{A}(r_1) = (Q_1, \Sigma, \Delta_1, 1, \{11, 12\})$
- $\mathcal{A}(r_2) = (Q_2, \Sigma, \Delta_2, 2, \{21, 22\})$

5 $\mathcal{A}(r_1 + r_2) = (Q, \Sigma, \Delta, 0, F)$
  - $Q = Q_1 \cup Q_2 \cup \{0\}$
  - $\Delta = \Delta_1 \cup \Delta_2 \cup \{(0, \varepsilon, 1), (0, \varepsilon, 2)\}$
  - $F = \{11, 12, 21, 22\})$

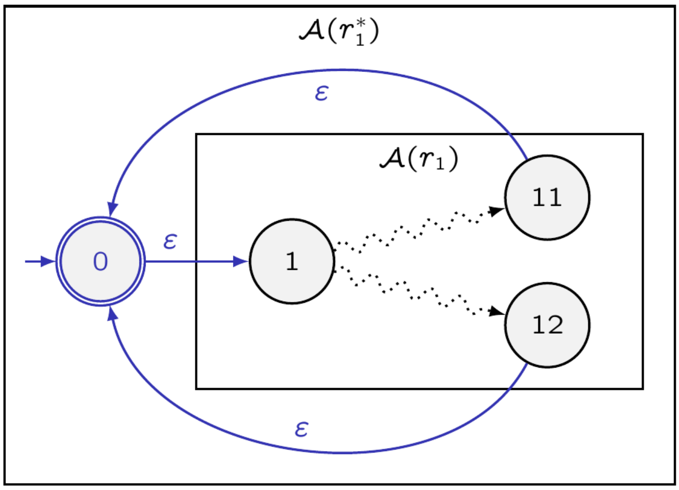
- Neuer Startzustand 0
- ε-Übergänge von 0 zu 1 und 2
- Endzustände 11, 12, 21 und 22



- $\mathcal{A}(r_1) = (Q_1, \Sigma, \Delta_1, 1, \{11, 12\})$

6 $\mathcal{A}(r_1^*) = (Q, \Sigma, \Delta, 0, \{0\})$
  - $Q = Q_1 \cup \{0\}$
  - $\Delta = \Delta_1 \cup \{(0, \varepsilon, 1), (11, \varepsilon, 0), (12, \varepsilon, 0)\}$

- Neuer Startzustand 0
- ε-Übergänge
  - von 0 zu 1
  - von 11 und 12 zu 0
- Endzustand 0



2.2.6 NEA zu DEA

Zuerst wird eine Transformationstabelle mit neuen Zuständen erstellt. Danach werden die Zustandsmengen als neue Zustände definiert und wenn die Menge einen Endzustand enthält, ist der neue Zustand ein Endzustand.

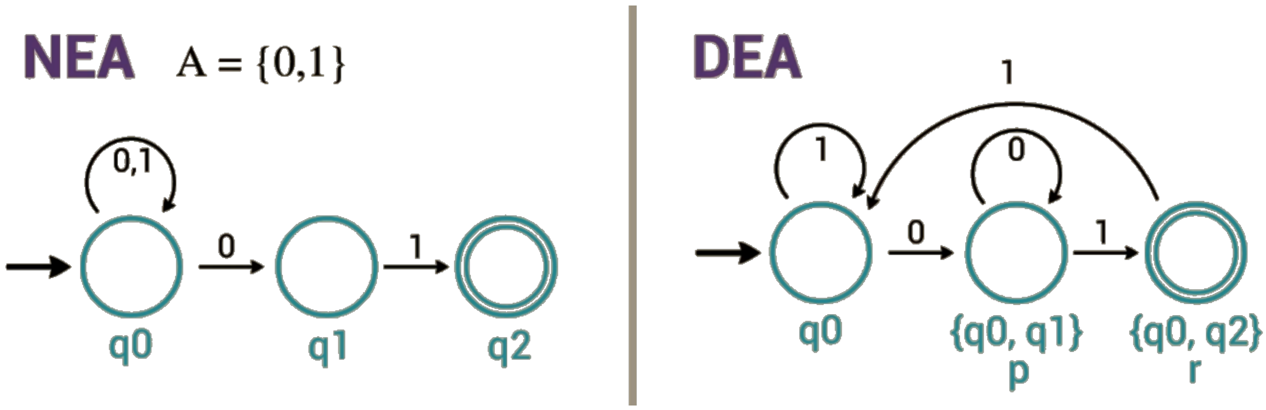
Übergangstabelle:

	0	1
q_0	$\{q_0, q_1\}$	q_0
q_1	$\varnothing$	q_2
$q_2{}^*$	$\varnothing$	$\varnothing$

Transformationstabelle:

	0	1
q_0	$\{q_0, q_1\}$	q_0
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}^*$
$\{q_0, q_2\}^*$	$\{q_0, q_1\}$	q_0

$\{q_0, q_1\}$ = p
$\{q_0, q_2\}^*$ = r



2.3 Minimierung

Bei der Minimierung werden Automaten vereinfacht. Hierzu werden zuerst alle unerreichbaren Zustände entfernt. Danach werden jeweils zwei Zustände miteinander verglichen. Falls einer der Zustände ein Endzustand ist und der andere nicht, wird diese Zelle in der Tabelle als unterscheidbar markeirt(im Bsp. 0). Falls dies nicht der Fall sein sollte, werden sich die Übergänge der zwei Zustände angesehen. Wenn die Übergänge der beiden Zustände jeweils auf die gleichen Folgezustände zeigen, sind diese nicht unterscheidbar und können am Ende zusammengefasst werden(Im Bsp.: X oder *leer*). Andernfalls sind diese ebenfalls unterscheidbar(im Bsp.: 1).

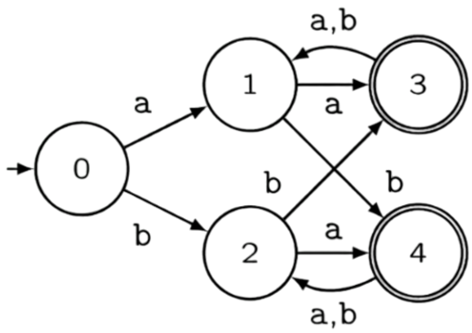
3 Teste Übergänge von jedem Zustandspaar für jeden Buchstaben

1 $\delta(0, \mathtt{a}) = 1; \delta(1, \mathtt{a}) = 3; (1, 3) \in U \rightsquigarrow (0, 1), (1, 0) \in U$

2 $\delta(0, \mathtt{a}) = 1; \delta(2, \mathtt{a}) = 4; (1, 4) \in U \rightsquigarrow (0, 2), (2, 0) \in U$

3 $\delta(1, \mathtt{a}) = 3; \delta(2, \mathtt{a}) = 4; (3, 4) \notin U$ (bisher)
$\delta(1, \mathtt{b}) = 4; \delta(2, \mathtt{b}) = 3; (4, 3) \notin U$ (bisher)

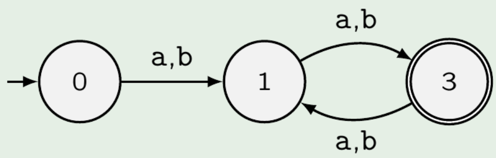
4 $\delta(3, \mathtt{a}) = 1; \delta(4, \mathtt{a}) = 2; (1, 2) \notin U$ (bisher)
$\delta(3, \mathtt{b}) = 1; \delta(4, \mathtt{b}) = 2; (1, 2) \notin U$ (bisher)



	0	1	2	3	4
0	×	1	1	0	0
1	1	×		0	0
2	1		×	0	0
3	0	0	0	×	
4	0	0	0		×

Hier wurden die Übergänge für (1,0),(2,0),(2,1),(4,3) als unentscheidbar erkannt. Bei (1,0) führt der Übergang a beim Zustand 0 zu 1 und beim Zustand 1 zu 3, (1,3) ist aber schon markiert, deshalb wird in die Zelle eine eins geschrieben. Wenn der Übergang nicht schon markiert wurde kann eine 0 in die Zelle geschrieben werden.

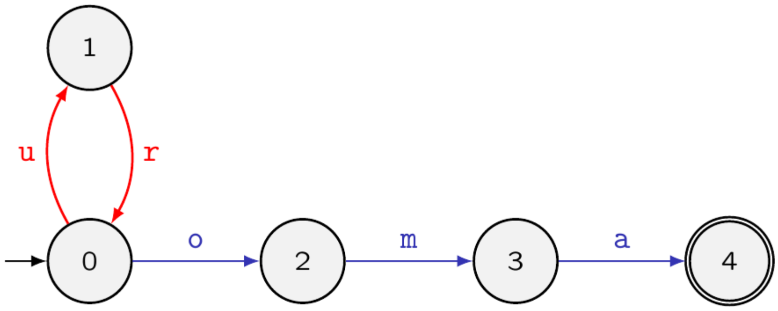
7 Vereinige Paare $(1, 2)$ und $(3, 4)$



Die Zustände bei denen ein X oder *leer* steht, können zusammengefasst werden. Damit ist die Minimierung abgeschlossen.

2.4 Nicht-reguläre Sprachen und das Pumping-Lemma

1. Das gesuchte Wort s besteht aus **Prolog** u, **Zyklus** v und **Epilog** w
2. Der Zyklus hat mindestest die Länge 1: $v \neq \epsilon$
3. Prolog und Zyklus zusammen haben höchstens die Länge k. Mit k $\leq |s|$
4. Eine beliebige Anzahl von Zyklus-Durchläufen erzeugt ein Wort der Sprache L: $s = uv^h w \in L$
5. $|uv| \leq k$



- $\mathcal{C}$ hat 5 Zustände — $k = 5$
- uroma hat 5 Buchstaben — $s = \texttt{uroma}$
- Es gibt eine Zerlegung $s = u \cdot v \cdot w$ — $u = \varepsilon \quad v = \texttt{ur} \quad w = \texttt{oma}$
- so dass $|v| \neq \varepsilon$ — $v = \texttt{ur}$
- und $|u \cdot v| \leq k$ — $|\varepsilon \cdot \texttt{ur}| = 2 \leq 5$
- und $\forall h \in \mathbb{N}(u \cdot v^h \cdot w \in \mathcal{L}(\mathcal{C}))$ — $(\texttt{ur})^*\texttt{oma} \subseteq \mathcal{L}(\mathcal{C})$

Hier ein anders Beispiel:

$$L = \{a^n b^m \mid n < m\}$$

Das Wort ist damit $a^n b^m$, dieses Word muss in u, v und w aufgeteilt werden um das Pumping Lemma anzuwenden.
$x = a^n b^{n+1} \in L$
$u = a^{n-k}$
$v = a^k$
$w = b^m$
Wir teilen u, v und w so auf, dass wir nun das Pumping Lemma anwenden können, wobei k>0 ist.

$$x = a^{n-k}(a^k)^i b^{n+1}$$

Laut dem Pumping Lemma können wir jetzt k beliebig wählen um das erzeugte Wort sollte immer noch ein Teil der Sprache L sein. i steht dabei für die Iterationen des Zyklus.
Also wählen wir $i = 3$ und setzen in die Gleichungen ein:

$$x = a^{n+k(-1+3)} b^{n+1}$$

$$x = a^{n+2k} b^{n+1}$$

Weil $k > 0$ wissen wir, das das $x \notin L$ und somit auch dass **L keine reguläre Sprache ist!**

2.5 Eigenschaften regulärer Sprachen

Eine Formale Sprache L ist regulär, wenn es einen **regulären Ausdruck**, einen **NEA** oder einen **DEA** gibt.

2.5.1 Leerheitsproblem

1. Startzustand als erreichbar markieren
2. Markiere iterativ alle erreichbaren Zustände als erreichbar

3. Stoppe, wenn ein Endzustand erreicht wurde oder wenn keine neuen erreichbaren Zustände gefunden werden
4. Falls ein Endzustand erreicht wurde: Ausgabe „nicht leer“
5. Sonst: Ausgabe „leer“ $\rightarrow$ Leerheitsproblem erfüllt!

2.5.2 Wortproblem

Man simuliert den Lauf von A auf ein Wort w. Dass heißt man versucht vom Startzustand auf das Wort w zu kommen.

2.5.3 Äquivalenzproblem

Herausfinden, ob zwei RA's r_1 und r_2 gleich sind.

1. Zwei NEA's erstellen(A_1, A_2)
2. NEA's in DEA's transformieren(D_1, D_2)
3. DEA's Minimieren (M_1, M_2)
4. Zustände von M_1 und M_2 umbennen.

Wenn $M_1 \equiv M_2$, dann gilt auch $r_1 \equiv r_2$

2.5.4 Endlichkeitsproblem

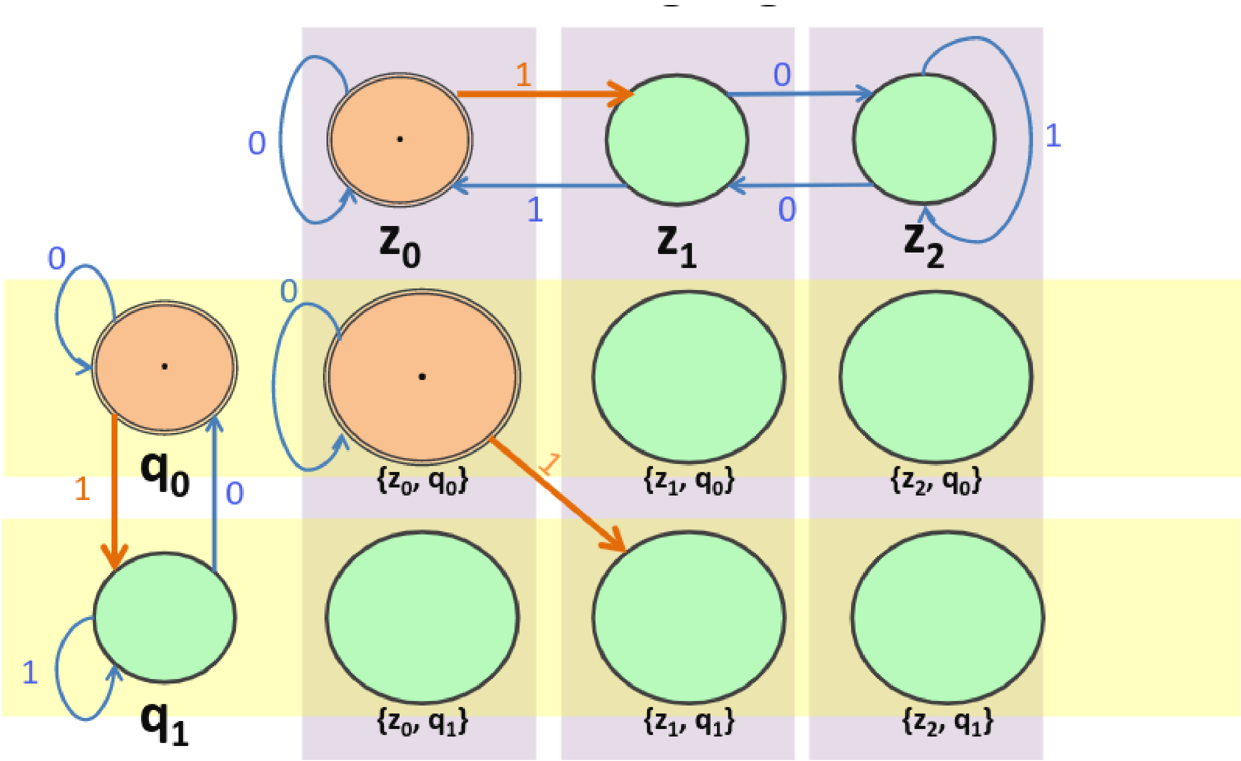
Ist das Problem, zu testen ob eine Sprache endlich ist.

1. Markiere iterativ alle vom Startzustand aus erreichbaren Zustände als erreichbar.
2. Markiere iterativ alle Zustände, von denen aus ein $q \in$ Endzustände F erreichbar ist, als terminierend
3. Sei A_r der Automat, der nur die erreichbaren und terminierenden Zustände von A enthält (gleiche Übergänge)

Ist A_r zyklisch folgt, dass $\mathcal{L}(A)$ unendlich ist

2.5.5 Produktautomaten

Kreuzprodukt der Zustände von A_1 und A_2 erstellen.
Für jeden Zustand die dazugehörigen Übergänge betrachten.
Dabei wird zum Beispiel für den Zustand (z_0, q_0) überprüft zu welchem Zustand der Übergang 1 führt. Der Zustand z_0 führt mit dem Übergang 1 zum Zustand z_1 und q_0 zu q_1, dass heißt das der Zustand (z_0, q_0) mit dem Übergang 1 zum Zustand (z_1, q_1) führt.



3 Chomsky Grammatiken und kontextfreie Sprachen

Gramatiken erzeugen formale Sprachen und sind auch für mächtigere Sprachklassen ausgelegt.

G=$(N, \sum, P, S)$ mit:

N Nichtterminalsymbole. Diese können für Regeln verwendet werden, aber dürfen nicht selbst im abgeleiteten Wort stehen.

P Ableitungsregeln. Bsp.: P=$\{S \rightarrow Aa|\varepsilon, A \rightarrow a\}$

S Startsymbol (ist Nicht-Terminal)

3.1 Chomsky-Hierarchie

3.1.1 Typ0 unbeschränkt

Jede Chromsky-Grammatik ist vom Typ 0.

3.1.2 Typ1 Monoton

$\alpha \rightarrow \beta$ mit $|\alpha| \leq |\beta|$ und Ausnahme Startsymbol $S \rightarrow \varepsilon$, wenn S auf keiner rechten Seite ist.
$\rightarrow$Regeln ersetzen nur einzelne NTS!

3.1.3 Typ2 Kontextfreie

A $\rightarrow \beta$ mit A $\in$ N und $\beta \in V^*$

3.1.4 Typ3 Rechtsregulär/-linear

A $\rightarrow$ cB mit A $\in$ N; B $\in$ N $\cup$ $\{\varepsilon\}$; c $\in$ $\sum \cup \{\varepsilon\}$

3.1.5 DEA zu RLG

A = (Q, $\sum$, δ, q_0, F) $\Rightarrow$ **G** = (N, $\sum$, P, S)
mit: **N** = Q, **S** = q_0 und **P** = $\{p \rightarrow cq | \delta(p, c) = q\} \cup \{p \rightarrow \epsilon | p \in F\}$
Bsp.: Kommt man mit a von Zustand 0 zu Endzustand 1: P=$\{0 \rightarrow a1, 1 \rightarrow \epsilon\}$

3.1.6 RLG zu NEA

G = (N, $\sum$, P, S) $\Rightarrow$ **A** = (Q, $\sum$, δ, q_0, F)
mit: **Q** = N $\cup \{f \mid f \notin N\}$, **F** = $\{f\}$, q_0 = S und
Δ = $\{(A, c, B) | A \rightarrow cB \in P\} \cup$
$\{(A, c, f) | A \rightarrow c \in P\} \cup$
$\{(A, \epsilon, B) | A \rightarrow B \in P\} \cup$
$\{(A, \epsilon, f) | A \rightarrow \epsilon \in P\}$
Prinzip: Regeln umwandeln und Endzustände f hinzufügen.

3.1.7 Kellerautomaten

Endlicher Automat mit unendlichem Stack, auf welchem nur der oberste/neueste Buchstabe gelesen, »gepusht« oder »gepopt« werden kann.
Akzeptanzbedingung: Leerer Stack und Wort zu Ende.
Deterministisch, wenn in jeder Konfiguration nur **eine** Folgkonfiguration möglich ist.
Syntax: A=(Q, Σ, Γ, Δ, q_0, Z)

Q Zustände

$\sum$ Alphabet

Γ Stack-Alphabet

q_0 Startzustand

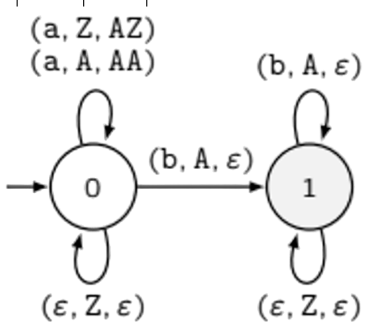
Z Startstacksymbol

Δ Regeln in Tabellenform $\Rightarrow$

Δ-Bsp.:

Q	$\sum$	Γ	Γ^*	Q
(0,	a,	A,	AA,	0)
(0,	b,	A,	ϵ,	1)
		...		

Im Graph Δ-Regeln ohne Zustände angeben:



Run/Konfigurationsfolge: 3er Tupel: (q, [Stack], Wort)

3.2 Cocke-Younger-Kasami(CYK)-Algorithmus

Entscheidet Wortproblem für kontextfreie Grammatiken in CNF. Bsp.:

$w = \text{abacba}$

Regeln:

$$\begin{aligned} S &\rightarrow a \\ B &\rightarrow b \\ B &\rightarrow c \\ S &\rightarrow SA \\ A &\rightarrow BS \\ B &\rightarrow BB \\ B &\rightarrow BS \end{aligned}$$

$i \setminus j$	1	2	3	4	5	6
1	S	{}	S	{}	{}	S
2		B	A, B	B	B	A, B
3			S	{}	{}	S
4				B	B	A, B
5					B	A, B
6						S
$w =$	a	b	a	c	b	a

Umgekehrte Regeln:

$$\begin{aligned} SA &\leftarrow S \\ BS &\leftarrow A, B \\ BB &\leftarrow B \end{aligned}$$

1. Wort unter Tabelle schreiben
2. Umgekehrte Regeln bilden (optional)
3. Herleitungen in Tabellen-Diagonale eintragen
4. Herleitungen der Teilwörter als Menge eintragen (alle Möglichkeiten)

3.3 Chomskynormalform CNF

Zur Entscheidung des Wortproblems. Nur Regeln mit Syntax:
$A, B, C \in Nichtterminalsymbole \ \wedge \ a \in Terminalsymbole$

- $A \rightarrow BC$
- $A \rightarrow a$
- $S \rightarrow \epsilon$, wenn S auf keiner Rechten Seite ist.

Vorgehen:

1. Epsilon-Regeln entfernen
   (a) Erstelle Liste L mit $A \rightarrow \epsilon$-Regeln und allen Regeln, die auf Nichtterminalsymbole in L zeigen.
   (b) Ergänze rechte Seite der Regeln mit sich selbst mit eingesetztem ϵ für Nichtterminalsymbole $\in$ L
   (c) Wenn $S \in L$ füge Regel $S_0 \rightarrow S|\epsilon$ ein
2. Kettenregel entfernen
   (a) Erstelle Listen für Kettenregeln ausgehend von jeweiligen Nichtterminalen
   (b) Ergänze Regeln mit allen Kettenregeln
   (c) Regeln kürzen
3. überflüssige Symbole entfernen
4. Einzelne Nichtterminalsymbole auf rechter Seite ersetzen
5. Rechte Seite mit Hilfssymbolen kürzen

3.4 Eigenschaften kontextfreier Sprachen

3.4.1 Pumping-Lemma 2

Für jedes $s \in L$ mit $|s| \geq k$ gibt es eine Zerlegung $s = u \cdot v \cdot w \cdot x \cdot y$

1. $vx \neq \epsilon$
2. $|vwx| \leq k$
3. $u \cdot v^h \cdot w \cdot x^h \cdot y \in L$ für alle $h \in \mathbb{N}$

Wie im normalen Pumping Lemma wird ein Wort der Sprache in mehrere Teile aufgespalten, wobei v und x aufgepumpt werden, um zu zeigen, dass sie die Regeln der Sprache verletzen, wodurch gezeigt wird, dass die Sprache nicht kontextfrei ist.
Mit einer kontextfreien Sprache können Vereinigungen, Konkatenation und Kleene-Stern verwendet werden. Durchschnitt und Komplement aber nicht.
Das Äquivalenzproblem ist aber unentscheidbar.

3.5 Regularität beweisen

Um die Regularität einer Grammatik zu beweisen, kann man aus der Grammatik oder dem regulären Ausdruck einen DEA oder NEA bilden, der die gleiche Sprache beschreibt. Falls man die Regularität widerlegen will, muss man das Pumping-Lemma anwenden!

4 Turing Maschine

4.1 Turing Maschine mit einem endlosem Einleseband

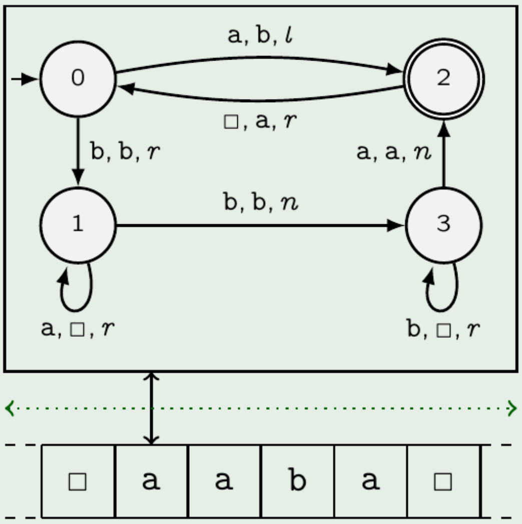
Terminiert wenn Endzustand erreicht und Einleseband nicht verschiebbar ist.
M=(Q, $\sum, \Gamma, \Delta, q_0$, F) mit:

Γ Vereinigung aus mindestens Blank-Symbol und Terminalsymbole: $\Gamma \supseteq \sum \cup \{\square\}$

Δ Übergangsrelationen Syntax: IST-Zustand, IST-Inhalt, Neuer-Inhalt, Verschiebung, Neuer-Zustand
Bsp.: $0, \quad \square, \quad a, \quad r, \quad 1$

F Endzustände

Run/Konfigurationsfolge: startet mit » q_0w «, mit w $\in \sum$* und akzeptiert, wenn $q \in F$, bzw. q sich im Endzustand befindet!



Konfigurationen:

1. 0aaba
2. 2□baba (akzeptierend)
3. a0baba
4. ab1aba
5. ab□1ba
6. ab□3ba
7. ab□□3a
8. ab□□2a (akzeptierend, Stop)

4.2 Mehrband-Turingmaschine

Wie eine normale Turingmaschine, mit dem Unterschied, dass es mehere Bänder gibt.

$\mathcal{M}_2 = (Q, \Sigma_{\text{abc}}, \Gamma, \Delta, \text{start}, \{\text{f}\})$ mit

- $Q = \{\text{start, reada, readb, readc, f}\}$
- $\Gamma = \Sigma_{\text{abc}} \cup \{\square\}$
- Δ siehe Tabelle

Vorteile:

- nur $\mathcal{O}(|w|)$ Schritte
- einfachere Übergangsrelation
- keine zusätzlichen Band-Symbole
- kein Abschluss-Check

start	$\begin{pmatrix}\square\\\square\end{pmatrix}$	$\begin{pmatrix}\square\\\square\end{pmatrix}$	$\begin{pmatrix}n\\n\end{pmatrix}$	f
start	$\begin{pmatrix}a\\\square\end{pmatrix}$	$\begin{pmatrix}a\\a\end{pmatrix}$	$\begin{pmatrix}r\\r\end{pmatrix}$	reada
reada	$\begin{pmatrix}a\\\square\end{pmatrix}$	$\begin{pmatrix}a\\a\end{pmatrix}$	$\begin{pmatrix}r\\r\end{pmatrix}$	reada
reada	$\begin{pmatrix}b\\\square\end{pmatrix}$	$\begin{pmatrix}b\\\square\end{pmatrix}$	$\begin{pmatrix}n\\l\end{pmatrix}$	readb
readb	$\begin{pmatrix}b\\a\end{pmatrix}$	$\begin{pmatrix}b\\b\end{pmatrix}$	$\begin{pmatrix}r\\l\end{pmatrix}$	readb
readb	$\begin{pmatrix}c\\\square\end{pmatrix}$	$\begin{pmatrix}c\\\square\end{pmatrix}$	$\begin{pmatrix}n\\r\end{pmatrix}$	readc
readc	$\begin{pmatrix}c\\b\end{pmatrix}$	$\begin{pmatrix}c\\c\end{pmatrix}$	$\begin{pmatrix}r\\r\end{pmatrix}$	readc
readc	$\begin{pmatrix}\square\\\square\end{pmatrix}$	$\begin{pmatrix}\square\\\square\end{pmatrix}$	$\begin{pmatrix}n\\n\end{pmatrix}$	f

Die Maschinenmodelle „Turing-Maschine“ und „ k-Band-Turing-Maschine“ sind äquivalent.

- Nichtdeterministische Turingmaschinen
  - können durch eine deterministische 2-Band-TM simuliert werden
  - beschreiben dieselbe Sprachklasse wie 1-Band-Turingmaschinen

4.3 Unbeschränkte Grammatiken

Gegeben: Grammatik $G = (N, \Sigma, P, S)$
Verwende nicht-deterministische 2-Band-TM:

- Band 1 speichert Eingabewort w
- Band 2 simuliert von S ausgehende Ableitungen gemäß P
- Ablauf:
  1. wähle (nicht-deterministisch) Position p auf B2
  2. wenn das auf p beginnende Wort zu einer Regel $\alpha \to \beta$ passt
     - verschiebe Bandinhalt wenn nötig
     - ersetze α durch β
  3. vergleiche B2 mit B1
     - wenn gleicher Inhalt, gehe in akzeptierende Stopkonfiguration
     - sonst weiter bei 1

Zusammenfassend: Turing Maschinen erkennen Sprachen die durch unbegränzte Grammatilken erzeugt werden. Im ersten Band wird das Eingabewort gespeichert, während im zweiten Band versucht wird das gleiche Wort anhand der Regeln zu erstellen. Wenn das Wort im Band1 und Band2 gleich sind, dann wird die eingabe Akzeptiert.

4.4 Linear beschränkter Automat

Touring Automat mit beschränktem Band durch Start- und Endsymbol (Band: "$> abaab <$")

- Monotone Grammatiken: Keine kürzenden Regeln
  - Ableitung von w enthält keine Wörter, die länger als w sind
- LBA: Turingmaschine, deren Band auf die Länge von w beschränkt ist
  - Marker kennzeichnen Grenzen von w
  - Schreib-/Lesekopf darf Marker nicht passieren oder überschreiben

...	>	i	n	p	u	t	<	...

- Mehrband-LBA möglich

Definition 4.20 (LBA)

Ein linear beschränkter Automat $\mathcal{A}$ ist eine Turing-Maschine $(Q, \Sigma, \Gamma, \Delta, q_0, F)$ so dass

- $\{>, <\} \subset \Gamma \setminus \Sigma$ gilt und
- Δ Übergänge mit Markern nur in der Form $(q, >, >, r, q')$ oder $(q, <, <, l, q')$ enthält.

Die von $\mathcal{A}$ akzeptierte Sprache ist die Menge aller Wörter $w \in \Sigma^*$, für die von der Startkonfiguration $>q_0w<$ aus eine akzeptierende Stopkonfiguration erreicht wird.

5 Entscheidbarkeit

Eine Menge S heißt entscheidbar, wenn eine deterministische Turing-Maschine M existiert, die eine Eingabe w ...

- ...akzeptiert, wenn w $\in$ S gilt.
- ...verwirft, wenn w $\notin$ S gilt.

5.1 Unentscheidbarkeit des speziellen Wortproblems

Das Spezielle Wortproblem W ist die Menge aller Turing Maschinen die ihre sich selbst akzeptieren. Wenn eine Sprache L entscheidbar ist, dann auch ihr Komplement. Das heißt eine Codierung einer Turing Maschine, welche sich selbst als Eingabe akzeptiert gütlig ist, dann auch die Codierung (Opposite), welche dies nicht tut. Dies führt zu einen Paradox, denn wenn Opposite nun sich selbst nicht akzeptiert, muss sie sich selbst akzeptierten, denn Opposite ist die Codierung von Turing Maschinen, welche sich selbst nicht akzeptieren. (BRAIN FUCK)

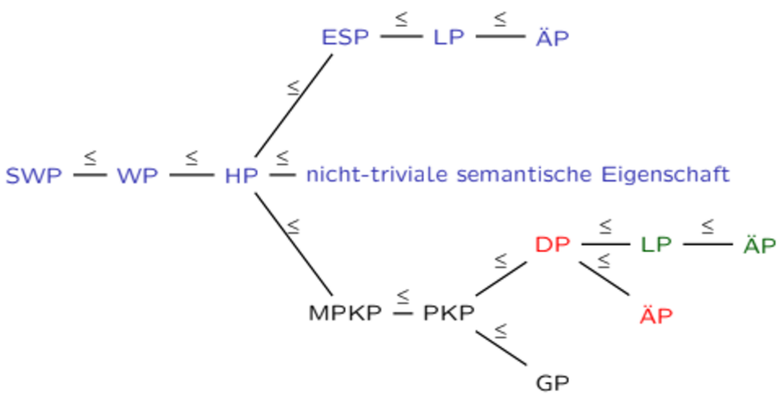
Turing-Maschine $\mathcal{M}$	ausgeführtes Programm
Gödelisierung $g(\mathcal{M})$	Programm als Binärdatei
akzeptieren	terminieren mit Rückgabewert 0
$\overline{W}$	Menge aller Programme, die sich selbst als Eingabe nicht akzeptieren
$\mathcal{M}_{\overline{W}}$	Programm *Opposite*, das genau die Programme akzeptiert, die sich selbst als Eingabe nicht akzeptieren

Angenommen, *Opposite* existiert.
Was macht *Opposite*, wenn es sich selbst als Eingabe erhält?

1. *Opposite* akzeptiert. Da *Opposite* genau die Programme akzeptiert, die sich selbst nicht akzeptieren, folgt, dass *Opposite* sich selbst nicht akzeptiert. $\lightning$
2. *Opposite* akzeptiert nicht. Da *Opposite* genau die Programme akzeptiert, die sich selbst nicht akzeptieren, folgt, dass *Opposite* sich selbst akzeptiert. $\lightning$

Also existiert *Opposite* nicht.

5.2 Reduktionsweise



Turing-Maschinen / rekursiv aufzählbare Sprachen
kontextsensitive Sprachen
kontextfreie Sprachen

5.3 Das PKP(Postsches Korrespondenzproblem) und weitere unentscheidbare Probleme

Endliche Folge an Wortpaaren, mit nichtleeren Wörtern, über endlichem Alphabet.
Syntax: P=$((l_1, r_1), (l_2, r_2), ...(l_n, r_n))$ **Lösung:** $l_{i_1} \cdot l_{i_2} \cdot ... \cdot l_{i_n} = r_{i_1} \cdot r_{i_2} \cdot ... \cdot r_{i_n}$

Sei $\Sigma = \Sigma_{ab}$.

1. $P_1 = ((a, aaa), (abaa, ab), (aab, b))$

   Lösung 1: $(2, 1)$
   - abaaa
   - abaaa

   Lösung 2: $(1, 3)$
   - aaab
   - aaab

2. $P_2 = ((ab, aba), (baa, aa), (aba, baa))$

   Keine Lösung:
   - Lösung muss mit Paar 1 beginnen
   - Danach nur Paar 3 möglich (beliebig oft)
   - Erstes Wort bleibt immer kürzer als zweites

   - ababaabaaba...
   - ababaabaabaa...

5.3.1 MPKP

Definition 5.34 (Modifiziertes PKP (MPKP))

Eine Instanz des MPKP ist definiert wie eine Instanz des PKP.
Eine Lösung des MPKP ist eine Indexfolge, die mit 1 beginnt.

Es zählen nur Lösungen, die mit dem ersten Wortpaar beginnen.

5.4 Semi-Entscheidbarkeit

Sei Σ ein Alphabet. Eine Sprache $L \subseteq \Sigma^*$ heißt semi-entscheidbar, wenn es eine DTM gibt, die mit Eingabe $w \in \Sigma^*$

- terminiert, wenn $w \in L$ gilt
- nicht terminiert sonst

5.5 Die universelle Turingmaschine

keine Ahnung?

5.6 Abschlusseigenschaften

keine Ahnung?

6 Berechenbarkeit

6.1 Turing Berechenbarkeit

keine Ahnung?

6.2 WHILE Programme

Variablen	$x_0, x_1, x_2, ...$
Konstanten	$0, 1, 2, ...$
Zuweisungsoperator	:=
Kompositionsoperator	;
Arithmetische Operatoren	+,-
Schlüsselwörter	**while, do, end**

while: solange die Variable nicht 0 ist
loop: While Schleife für Variable (for-Schleife in Java)

1. Wertzuweisung **immer mit Konstante hinten**

$$x_i := x_j + c$$

2. Komposition Sind P1 und P2 WHILE-Programme, dann auch

$$P_1; P_2$$

3. While-Schleife: Ist P ein WHILE-Programm und i 2 N, dann auch "while $x_i$do P end"

Beispiel 6.11 (LOOP-Schleife)

loop x_i **do** P **end**: Schleife mit fester Anzahl x_i von Durchläufen

$x_j := x_i + 0;$
while x_j **do**
    $x_j := x_j \dot{-} 1;$
    P
end

x_j wird sonst nicht benutzt

Beispiel 6.12 (IF-THEN-ELSE-Konstrukt)

if x_i **then** P_1 **else** P_2 **end**: Wenn $x_i \neq 0$ gilt, wird P_1 ausgeführt, sonst P_2

loop x_i **do**
    $x_j := x_l + 1$ // $x_i \neq 0 \rightsquigarrow x_j = 1$
end;
$x_k := x_l + 1;$
loop x_j **do**
    $x_k := x_l + 0$ // $x_j \neq 0 \rightsquigarrow x_k = 0$
end;
loop x_j **do**
    P_1 // $x_j \neq 0 \rightsquigarrow P_1$ wird ausgeführt
end;
loop x_k **do**
    P_2 // $x_k \neq 0 \rightsquigarrow P_2$ wird ausgeführt
end

x_j, x_k, x_l werden sonst nicht benutzt

Beispiel 6.16 (Variablen-Subtraktion)

$$s : \mathbb{N}^2 \to \mathbb{N} \text{ mit } s(x_1, x_2) = \begin{cases} x_1 - x_2 & \text{wenn } x_1 \geq x_2 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Die Funktion s ist WHILE-berechenbar:

while x_2 **do** // Solange $x_2 > 0$ gilt
    if x_1 **then** // Wenn $x_1 > 0$ gilt
        $x_1 := x_1 \dot{-} 1;$
        $x_2 := x_2 \dot{-} 1$ // dekrementiere x_1 und x_2 parallel
    end
end; // terminiert nicht, wenn $x_2 > x_1$ gilt
$x_0 := x_1 + 0$

6.3 Die Church-Turing-These

keine Ahnung?

7 Komplexität

7.1 Komplexitätsklassen

Die Komplexität einer Turing Maschine, bzw eines Algorithmusses wird in Schritten angegeben. Eine Turingmaschine M heißt f-zeitbeschränkt, wenn M für jede Eingabe w höchstens f(w) Schritte hat. Für DEA heißt diese TIME und für NEA NTIME. Diese beschreibt nicht die Komplexität eines speziellen Algorithmus, sondern des Problems, d.h. des besten Algorithmus.
P steht für polynomiell und NP für nichtdeterministisch polynomiell.

7.2 NP-Vollständigkeit

Keine Ahnung?