Inhaltsverzeichnis

Aufgabenlösungen ab S. 53!

Teil I Formale Sprachen und Automatentheorie

1. Allgemeines

1.1. Grundbegriffe

Im folgenden: Abstraktion von Alphabet, Wort, Grammatik und Sprache

Definition:

- Alphabet Σ: endliche, nicht-leere Menge von Zeichen
  Beispiel: $\Sigma=\{0,1\}$
- Wort ω über Σ: endliche Folge von Zeichen aus Σ
  Beispiel: $\omega=01011$
- Länge eines Wortes ω über Σ: $|\omega|$
  Beispiel: $|\omega|=5$
- Leeres Wort ϵ: enthält keine Zeichen, $|\epsilon|=0$
- Σ^*: Menge aller Worter über Σ (ist immer unendlich)
- Σ^+: Menge aller nicht-leeren Wörter über Σ
- Sei $a\in\Sigma: a^k := \underbrace{a\,a\ldots a}_{k-\text{mal}}, a^0 := \epsilon$
  Beispiel: $0^4=0000$
- Verkettung (Konkatenation) $\omega_1\cdot\omega_2$ zweier Wörter $\omega_1,\omega_2\in\Sigma^*$ ist das Wort, das durch Hintereinanderschreiben von ω_1 und ω_2 entsteht
  Beispiel: $01\cdot011=01011$

Definition: (Formale Sprache)

Sei Σ ein Alphabet. Eine formale Sprache L (über Σ) ist eine beliebige Teilmenge von Σ^*.

Präzisierung der formalen Sprache durch:

a) Auflisten aller Elemente von L (nur bei endlichem L)

b) Mathematische oder informelle Beschreibung einer Gesetzmäßigkeit

c) Endliche Erzeugungsverfahren (Grammatiken: Angabe von Regeln, die alle Wörter von L erzeugen).

d) Endliche Erkennungsverfahren (Automaten: Angabe eines Algorithmus, der entscheidet, ob $\omega\in L$).

1.2. Grammatik

Definition:

Eine Grammatik ist ein 4-Tupel $G=(V,\Sigma,R,S)$ mit

- V: Alphabet von Variablen (Nonterminale)
- Σ: Alphabet der Terminale
- R: einer endlichen Menge von Regeln (Produktionen)
  $\alpha \to \beta$
  mit $\alpha \in (V \cup \Sigma)^+, \beta \in (V \cup \Sigma)^*$
- $S \in V$: Startvariable

Beispiel:

$G_{ARI} := (\{S\}, \{+, *, a, (,)\}, R, S)$ mit
$R = \{ S \to S+S,$
$S \to S*S,$
$S \to (S),$
$S \to a \}$

G_{ARI} erzeugt korrekt geklammerte arithmetische Ausdrücke.

S
$S+S$
$S+(S)$
$S+(S*S)$
$\vdots$
$a+(a*a)$

Konvention:

kleine lateinische Buchstaben $(a, b, \ldots)$	Terminale
große lateinische Buchstaben $(A, B, \ldots)$	Variable
kleine griechische Buchstaben $(\alpha, \beta, \ldots)$	Wörter, bestehend aus Terminalen und Variablen

Regelmenge definiert dann Grammatik vollständig: Linke Seite der ersten Regel gibt Startvariable an.

Sei $G = (V, \Sigma, R, S)$ eine Grammatik. Wir definieren die Relation
$\alpha \Rightarrow_G \beta$
(α geht unter G in einem Schritt in β über), falls α, β die Form
$\alpha = \gamma_1 \tau \gamma_2$
$\beta = \gamma_1 \nu \gamma_2$
haben, und $\tau \to \nu \in R$.
Wenn die Grammatik klar ist $\to$ kurz: $\alpha \Rightarrow \beta$

$\underbrace{\gamma_1}\ \underbrace{\tau}\ \underbrace{\gamma_2}$
$s+(s)$
$\underbrace{s+(}\ \underbrace{s*s}\ \underbrace{)}$
$\gamma_1\ \ \nu\ \ \gamma_2$

$\underbrace{S}_{\tau} \to \underbrace{(S)}_{\nu}$ $\quad \gamma_1 = \epsilon \quad \gamma_2 = \epsilon$

$\alpha \Rightarrow_G^* \beta$ (in Worten: α geht unter G in endlich vielen Schritten in β über), wenn $\alpha = \beta$ oder Wörter $\omega_0, \omega_1, \ldots, \omega_n$ mit $\omega_0 = \alpha, \omega_n = \beta$ und $\omega_i \to_G \omega_{i+1}$ für $i = 0, \ldots, n-1$.

Wir nennen $S \Rightarrow_G^* \omega \in \Sigma^*$ eine Ableitung, und $S \Rightarrow \omega_1 \Rightarrow \ldots \Rightarrow \omega_n = \omega$ eine Ableitungsfolge für ω.

Beispiel: mit G_{ARI}

$S \Rightarrow S+S \Rightarrow S*S+S \Rightarrow S*S*S+S \Rightarrow S*S*(S)+S \Rightarrow S*S*(S+S)+S \Rightarrow S*S*(S+S)+S \Rightarrow \ldots$
$\Rightarrow a*a*(a+a)+a$
$S \Rightarrow^* a*a*(a+a)+a$

Beispiel: einfache deutsche Sätze

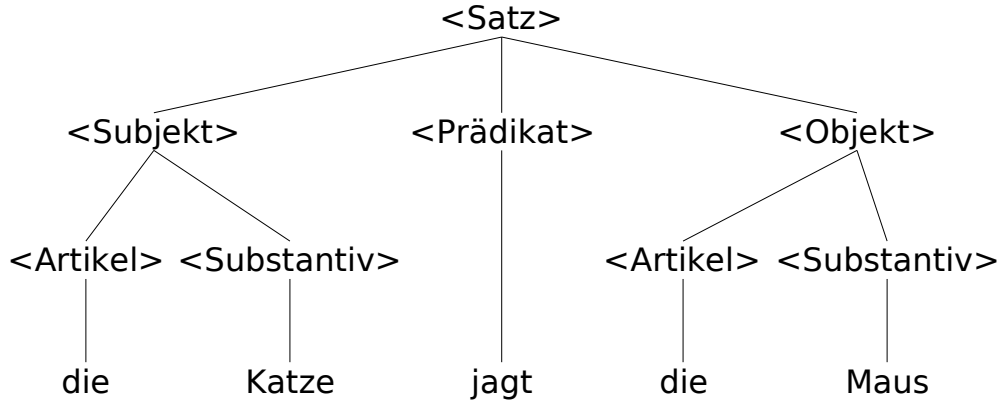
hier: eine Variable!

<Satz> → <Subjekt> <Prädikat> <Objekt>
<Subjekt> → <Artikel> <Substantiv>
<Artikel> → die
<Substantiv> → Katze
<Substantiv> → Maus
<Substantiv> → Ratte
<Prädikat> → jagt
<Prädikat> → frisst
<Objekt> → <Artikel> <Substantiv>
Spitze Klammern: Variablen; Startvariable: <Satz>

Durch diese Grammatik werden z.B. folgende Wörter (hier: Sätze) erzeugt:
die Katze frißt die Maus
die Maus jagt die Ratte usw.

Anschauliche Darstellung einer Ableitung: Ableitungs- oder Syntaxbaum

Wurzel: Startvariable
Elternknoten: linke Seite der Regel
Kinder: Objekte auf der rechten Seite der Regel



Definition: (von Grammatik erzeugte Sprache)

Sei $G=(V,\Sigma,R,S)$ Grammatik. Die von G erzeugte Sprache ist die Menge $L(G):=\{\omega\in\Sigma^*|S\Rightarrow_G^*\omega\}$.

G_{ARI}: $S\to(S)$ $S\to S*S$
$S\to S+S$ $S\to a$

Es gilt z.B.

$$a*a*(a+a)+a\in L(G_{ARI})$$

und

$$a(a+a)\notin L(G_{ARI})$$

1.3. Chomsky-Hierarchie

Noam Chomsky (1956)

Definition: (CHOMSKY-GRAMMATIK VOM TYP i)

Eine Grammatik $G=(V,\Sigma,R,S)$ ist automatisch vom Typ 0 (rekursiv aufzählbar). Keine Einschränkungen für die Regeln.

Sie ist weiter vom Typ

1 (Kontextsensitiv) :⇔ alle Regeln sind vom Typ

$$\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$$ Kontext

mit $A \in V$; $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$; $\beta \in (V \cup \Sigma)^+$.

2 (Kontextfrei) :⇔ alle Regeln sind vom Typ

$$A \to \alpha$$

mit $A \in V$; $\alpha \in (V \cup \Sigma)^+$ auf der linken Seite steht nur eine Variable!

3 (regulär) :⇔ alle Regeln sind vom Typ

<int>→0|1
0<int>

$$A \to aB$$

oder

$$A \to a$$

mit $A, B \in V$; $a \in \Sigma$ („Rechtslinearität“)

Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ i ($i=0,1,2,3$), wenn eine Grammatik G vom Typ i existiert mit $L(G)=L$.

Soll für eine Grammatik vom Typ 1, 2, 3 gelten: $\epsilon \in L(G)$, so wird Regel $S \to \epsilon$ zugelassen, S darf dann aber auf keiner rechten Seite mehr auftauchen („ϵ-Freiheit“).

Satz:

Jede Typ-i-Grammatik ($i=1,2,3$) ist Typ-(i-1)-Grammatik.

Selbiges gilt für Sprachen.

Beispiele:

Typ 3: $S \to 0S$, $S \to 0$ $\quad \Sigma=\{0\},\ V=\{S\}$

$$S \Rightarrow 0S \Rightarrow 00S \Rightarrow \ldots \Rightarrow 0^{n-1}S \Rightarrow 0^n$$

Es ist $L(G)=\{0^n \mid n \geq 1\}$

Typ 2: $S \to 0S1$, $S \to 01$ $\quad \Sigma=\{0,1\},\ V=\{S\}$

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow \ldots \Rightarrow 0^{n-1}S1^{n-1} \Rightarrow 0^n 1^n$$

$L(G)=\{0^n 1^n \mid n \geq 1\}$

Typ 0:

$S \to 0SBC$ 1
$S \to 0BC$ 2
$CB \to BC$ 3 ← nicht kontextsensitiv
$0B \to 01$ 4
$1B \to 11$ 5
$1C \to 12$ 6
$2C \to 22$ 7

$\Sigma = \{0,1,2\}$, $V = \{S,B,C\}$
S Startvariable

Nicht sofort offensichtlich: $L(G) = \{0^n 1^n 2^n | n \geq 1\}$

- Regeln 1, 2: $0^n (BC)^n$ aufbauen
- Regel 3: $0^n B^n C^n$
- Regeln 4, 5: in $0^n 1^n C^n$ umwandeln
- Regeln 6, 7: in $0^n 1^n 2^n$ umwandeln

Definition:

Eine Grammatik G heißt beschränkt, wenn sie keine verkürzenden Regeln enthält. D.h. aus $\alpha \Rightarrow_G \beta$ folgt $|\beta| \geq |\alpha|$.

Kontextsensitive Grammatiken sind beschränkt.

Umgekehrt gilt:

Satz:

Zu einer beschränkten Grammatik G gibt es eine eine äquivalente kontextsensitive Grammatik G' mit $L(G) = L(G')$.

$\mathcal{L}_i$ bezeichne die Menge aller Sprachen vom Typ i.

Satz:

$\mathcal{L}_3 \not\subseteq \mathcal{L}_2 \not\subseteq \mathcal{L}_1 \not\subseteq \mathcal{L}_0$

Echte Teilmengenrelation! → Chomsky-Hierarchie

- $L = \{0^n 1^n | n \geq 1\}$ ist Typ 2, nicht aber Typ 3
- $L' = \{0^n 1^n 2^n | n \geq 1\}$ ist Typ 1, nicht aber Typ 2 (wegen Beschränktheit)

1.4. Wortproblem

Definition: (Entscheidbarkeit)

Eine Aussage heißt entscheidbar, wenn es einen Algorithmus gibt, der nach endlich vielen Schritten die Antwort gibt, ob die Aussage wahr oder falsch ist.

Eine formale Sprache $L \subseteq \Sigma^*$ heißt entscheidbar, wenn für jedes $\omega \in \Sigma^*$ die Aussage „$\omega \in L$“ entscheidbar ist. „Wortproblem“

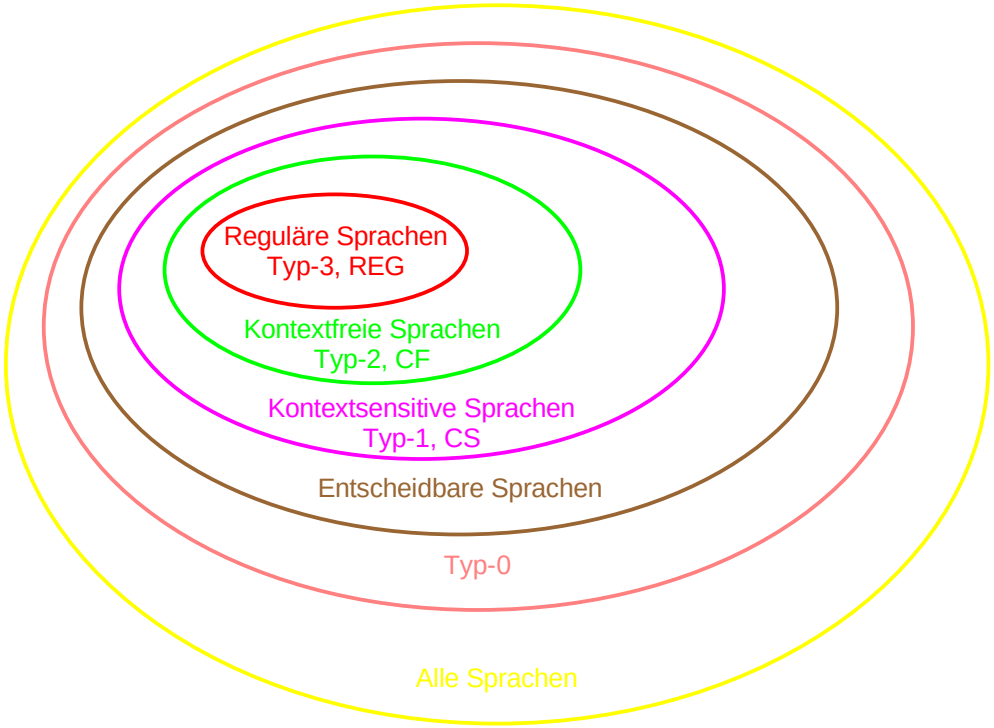
Satz:

Ist eine Sprache kontextsensitiv, dann ist sie entscheidbar.

Beweisidee:

Bei der Ableitung eines Wortes ω mit $|\omega| = n$ können alle „Zwischenergebnisse“ höchstens Länge n haben (keine verkürzenden Regeln). Grammatik hat endliche viele Regeln, systematisch alle Wörter mit Länge $\leq n$ erzeugen und schauen, ob ω drin ist.

Sprachehierarchie:



1.5. Backus-Naur-Form (BNF)

Verkürzte Notation für kontextfreie Grammatiken (häufig bei der Beschreibung von Programmiersprachen).

- Haben mehrere Regeln gleiche linke Seite:

  $A \to \beta_1$
  $A \to \beta_2$
  $\vdots$
  $A \to \beta_n$

  so schreibt man kurz

  $A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ ← alternativ rechte Seiten durch „|“ getrennt

- erweiterte BNF (EBNF)
  - optionale Einfügung

    $A \to \alpha [\beta] \gamma$ β kann – muß aber nicht – zwischen α und γ eingefügt werden.

    steht für

    $A \to \alpha \beta \gamma$
    $A \to \alpha \gamma$

  - beliebig häufige Einfügung

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

    steht für die Regeln

$A \rightarrow \alpha \gamma$
$A \rightarrow \alpha B \gamma$
$B \rightarrow \beta B$
$B \rightarrow \beta$

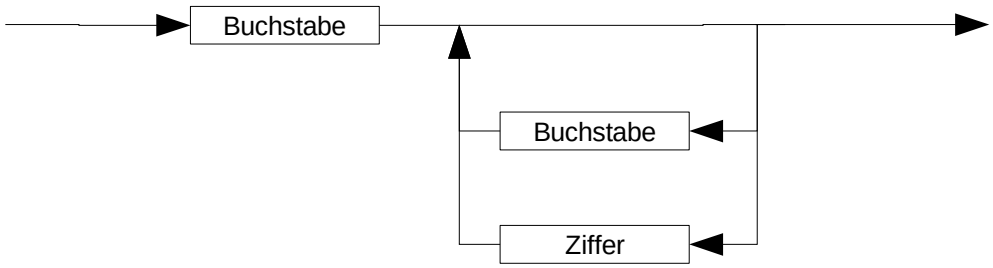
Original-(E)BNF: statt „→“ schreibt man „::=“, Variablen in spitzen Klammern.

Beispiel: Grammatik für zulässige Bezeichner

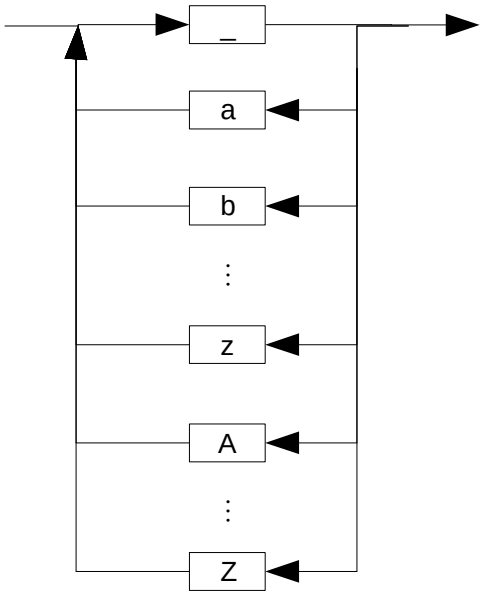
<Bezeichner> ::= <Buchstabe>|<Bezeichner> <Buchstabe>|<Bezeichner><Ziffer>
<Buchstabe> ::= _|a|b|...|z|A|...|Z
<Ziffer> ::= 0|...|9

Weiteres Mittel zur Beschreibung kontextfreier Sprachen: Syntaxdiagramme

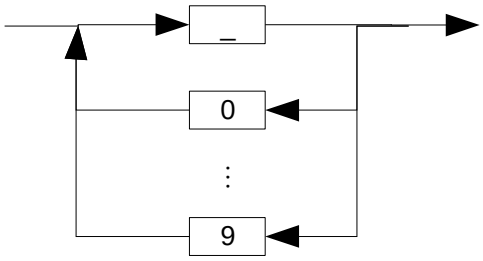
Bezeichner:



Buchstabe:



Ziffer:



2. Reguläre Sprachen

Im weiteren: Äquivalente Charakterisierungen und Eigenschaften von regulären (Typ-3)-Sprachen.

- Grammatiken <u>erzeugen</u> Wörter
- Automaten <u>akzeptieren / erkennen</u> Wörter

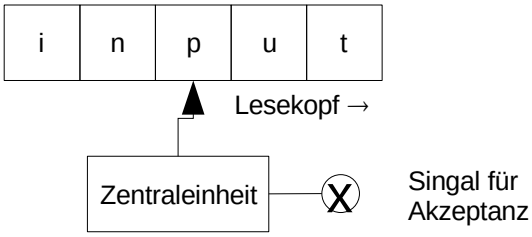
} Beide Konzepte definieren Sprache

2.1. Endliche Automaten

2.1.1. Deterministische endliche Automaten (DEA)

Bestandteile:

1. Zentraleinheit, die <u>endlich</u> viele Zustände annehmen kann
2. Eingabeband
3. Lesekopf, liest auf Eingabeband stehende Zeichen von links nach rechts



<u>Definition:</u> (Deterministischer Endlicher Automat (DEA))

Ein DEA $A=(Q,\Sigma,\delta,q_0,F)$; mit

- Q: einer <u>endlichen</u> Menge von Zuständen
- Σ: <u>Eingabealphabet</u>
- $\delta: Q\times\Sigma\to Q$ der <u>Übergangs</u>- oder Transitionsfunktion, die <u>jedem</u> Paar (a,q) mit $a\in\Sigma$, $q\in Q$ einen Folgezustand zuordnet.
- $q_0\in Q$: dem Startzustand
- $F\subseteq Q$: nichtleere Menge von <u>Endzuständen</u>.

Ein DEA A akzeptiert ein Wort $\omega := a_1\ldots a_n\in\Sigma^*$, wenn er ω von links nach rechts zeichenweise liest und dabei eine Folge von Zuständen $q_0, q_1, \ldots, q_n$ durchläuft, mit $q_i=\delta(q_{i-1},a_i)$ und $q_n\in F$.

Erweiterung von δ zu Wortfunktion $\delta^*: Q \times \Sigma^* \to Q$:

1. $\delta^*(q, \epsilon) = q$
2. $\delta^*(q, \underbrace{\omega}_{\in \Sigma^*} \underbrace{a}_{\in \Sigma}) = \delta(\delta^*(q, \omega), a)$

Meist wird auf Unterscheidung δ, δ^* verzichtet.

Definition:

Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein DEA. Dann ist die Sprache $L(A) := \{\omega \in \Sigma^* | \delta^*(q_0, \omega) \in F\}$ die Menge aller von A akzeptierten Wörter.

Beispiele:

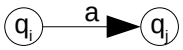
1. Sei $A = (Q, \Sigma, \delta, q_0, F)$ mit
   $Q := \{q_0, q_1, q_2, q_3\}$
   $\Sigma := \{0, 1\}$
   $F := \{q_0\}$
   Startzustand q_0
   δ tabellarisch:

Q / Σ	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

   Wird 0110101 akzeptiert?
   $q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_0 \xrightarrow{0} q_2 \xrightarrow{1} q_3 \xrightarrow{0} q_1 \xrightarrow{1} \underbrace{q_0}_{\in F}$

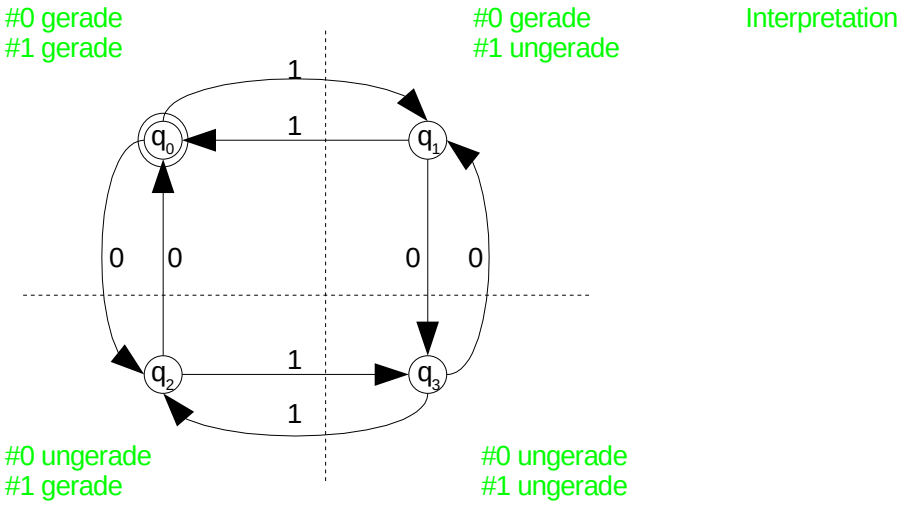
   Ja, wird akzeptiert.

   Grafische Darstellung: Zustandsübergangsdiagramme

   - Zustände: Knoten
   - Übergang $\delta(q_i, a) = q_j$ wird zu gerichteter Kante 

   - Startzustand wird mit „Einsprungpfeil“ versehen: 

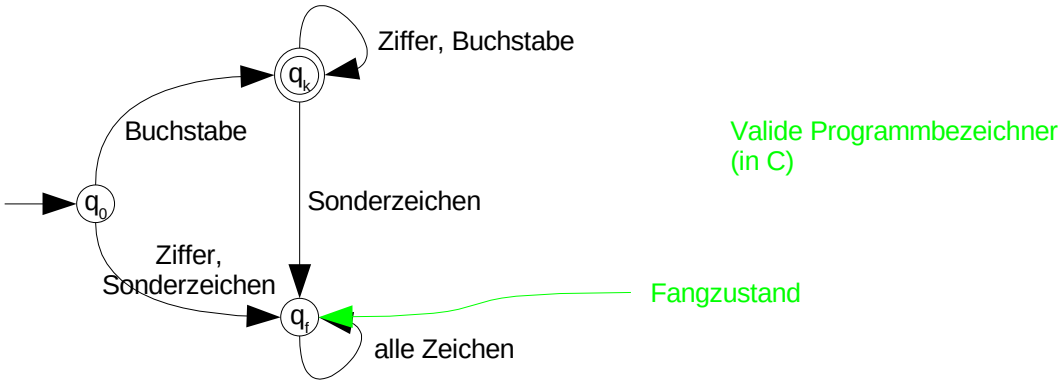
   - Endzustände doppelt umkringeln: 



$\rightarrow L(A)$: Menge aller Binärwörter, in denen Anzahl 0en und 1en gerade ist.

2. $A=(Q,\Sigma,\delta,q_0,F)$ mit
   $\Sigma :=$ Menge aller druckbaren ASCII-Zeichen (Buchstaben inkl. „_“, Ziffern und Sonderzeichen)
   $Q :=$ $\{q_0,q_k,q_f\}$
   Startzustand q_0
   $F:=\{q_k\}$
   δ tabellarisch:

Q / Σ	Buchstaben	Ziffern	Sonderzeichen
q_0	q_k	q_f	q_f
q_k	q_k	q_k	q_f
q_f	q_f	q_k	q_f



Satz: ZUSAMMENHANG DEA / REG

Jede durch einen DEA akzeptierte Sprache ist regulär.

Beweis: Konstruktiv

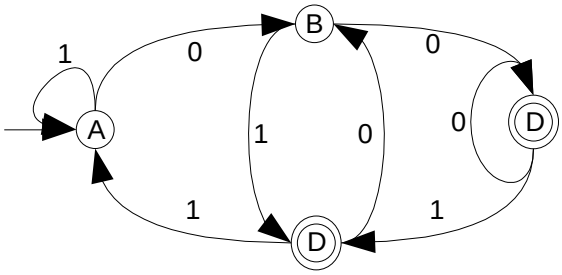
Sei $A=(Q,\Sigma,\delta,q_0,F)$ ein DEA. Wir erzeugen Typ-3-Grammatik $G=(Q,\Sigma,R,q_0)$, so daß $L(G)=L(A)$.

- Ist $\epsilon\in L(A)$, so enthält R die Regel

$q_0 \to \epsilon$

- Jedem Übergang $\delta(q_i, a) = q_j$ ordnen wir die Regel
  $q_i \to a\, q_j$
  zu, und falls $q_j \in F$ zusätzlich
  $q_i \to a$

Beispiel: DEA



Typ-3-Grammatik:

$A \to 1\, A$
$A \to 0\, A$
$B \to 0\, D$
$B \to 0$
$B \to 1\, C | 1$
$C \to 0\, B | 1\, A$
$D \to 0\, D | 0 | 1\, C | 1$

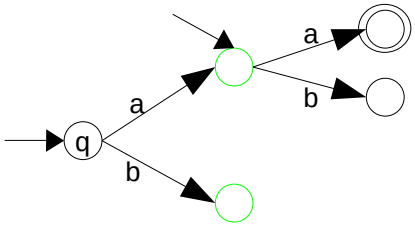
2.1.2. Nichtdeterministische endliche Automaten (NEA)

Definition: (NEA)

Ein NEA N ist ein 5-Tupel $(Q, \Sigma, \delta, Q_0, F)$ mit Q, Σ und F wie bei DEA, Q_0 einer Menge von Startzuständen und δ einer Abbildung von $Q \times \Sigma$ in die Potenzmenge von Q. $(\delta : Q \times \Sigma \to \wp(Q))$.

Ein Wort $\omega = a_1 \ldots a_n$ wird von NEA akzeptiert, wenn es mindestens eine Zustandfolge $q_0, q_1, \ldots q_n$ gibt mit $q_0 \in Q_0$, $q_i \in \delta(q_{i-1}, a_i)$ und $q_n \in F$.

- DEA: genau ein Folgezustand für ein Eingabezeichen
- NEA: (möglicherweise leere) Menge von Folgezuständen

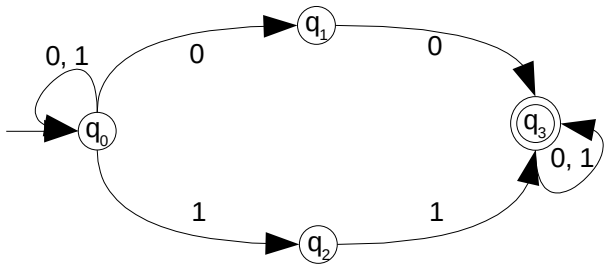


$\Sigma = \{a, b\}$

NEA kann von q aus mit a in einen der grünen Zustände wechseln, für b gibt es keinen Übergang.

Für gegebene Sprache läßt sich oft NEA leichter finden als DEA.

Beispiel:



NEA akzeptiert Wörter, die „00“ oder „11“ enthalten.

DEA sind Spezialfälle der NEA. Es gilt aber auch:

Satz: (Rabin, Scott)

Zu jeder von einem NEA akzeptierten Sprache gibt es einen DEA, der die selbe Sprache akzeptiert.

Beweisidee: (Potenzmengenkonstruktion)

Zu einem NEA $N=(Q,\Sigma,\delta,Q_0,F)$ wird ein DEA $A=(Q',\Sigma,\delta',q_0',F')$ konstruiert.

Zustandsmengen von Q werden Zustände in Q': $Q'\subseteq\wp(Q)$
Start:

$$q_0'=Q_0$$

N erreicht von $Q_0(=q_0')$ aus für jedes Eingabezeichen neue Zustandsmengen

→ neue Zustände von A.

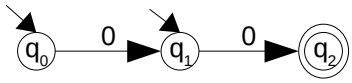
Für diese Zustandsmengen wiederholt man das Vorgehen, bis keine neuen Zustandsmengen mehr entstehen.

$$\delta'(\underbrace{\{q_1,\dots q_n\}}_{\text{Zustand DEA}},a):=\bigcup_{i=1}^{n}\delta(\underbrace{q_i}_{\text{Zustand NEA}},a)$$

F' bilden die Zustandsmengen in Q', die Zustände aus F enthalten.

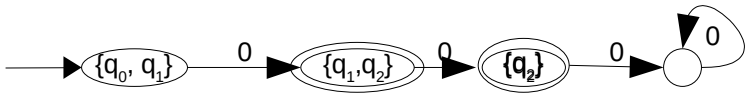
Beispiel 1:

$\Sigma=\{0\}$



δ' tabellarisch:

	Q' / Σ	0
Startzustand q'0	{q0,q1}	{q1,q2}
	{q1,q2}	{q2}
Endzustände	{q2}	

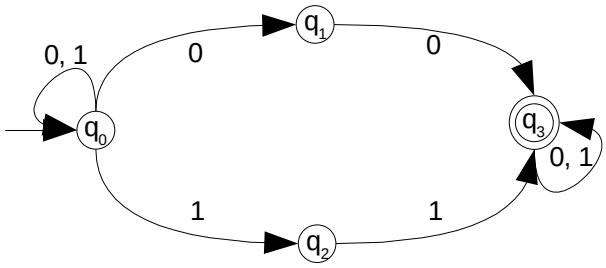


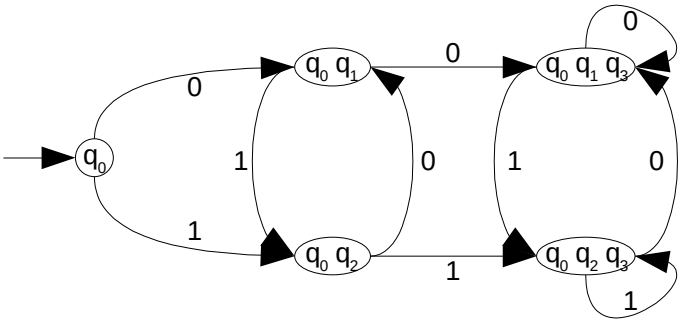
Beispiel 2:

Q' / Σ	0	1
q_0	$q_0\ q_1$	$q_0\ q_2$
$q_0\ q_1$	$q_0\ q_1\ q_2$	$q_0\ q_2$
$q_0\ q_2$	$q_0\ q_1$	$q_0\ q_2\ q_3$
$q_0\ q_1\ q_3$	$q_0\ q_1\ q_3$	$q_0\ q_2\ q_3$
$q_0\ q_2\ q_3$	$q_0\ q_1\ q_3$	$q_0\ q_2\ q_3$

F'

↑
kurz für:
$\{q_0, q_1, q_3\}$





Endliche Automaten: DEA oder NEA

Die von endlichen Automaten akzeptierten Sprachen sind genau die regulären Sprachen.

Satz:

Zu jeder regulären Grammatik G gibt es einen NEA N, der die von G erzeugte Sprache akzeptiert, d.h. $L(G)=L(N)$.

Beweisidee:

$G=(V,\Sigma,R,S)$ gegeben

Variablen werden zu Zuständen des NEA. Zusätzlich führt man einen Endzustand E ein.

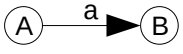
Aus einer Regel

$A \to a\,B$

leitet man Transition

$B \in \delta(A,a)$

ab,

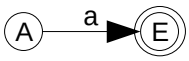


aus

$A \to a$

Transition

$E \in \delta(A,a)$

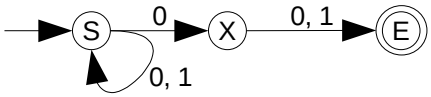


Beispiel:

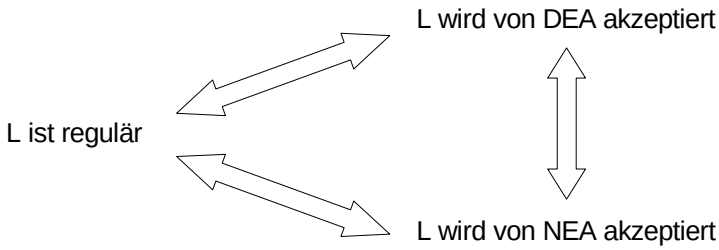
G: $S \to 0S|1S|0X$
$X \to 0|1$

Binärwörter mit 0 an vorletzter Stelle

NEA: $S \in \delta(S,0)$
$S \in \delta(S,1)$
$X \in \delta(S,0)$
$E \in \delta(X,0)$
$E \in \delta(X,1)$



Daraus folgt für formale Sprache $L \in \Sigma^*$:

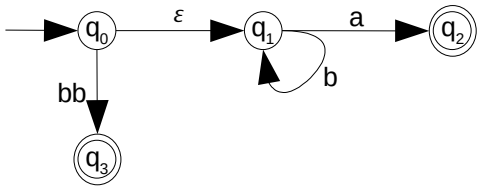


2.2. NEA mit ϵ-Kanten

Bei ϵ-NEA dürfen Kanten mit ϵ, aber auch mit Wörtern markiert sein.

ϵ-NEA kann

- in einem Schritt ein ganzes Wort verarbeiten
- Zustandsübergänge machen, ohne Zeichen zu verarbeiten



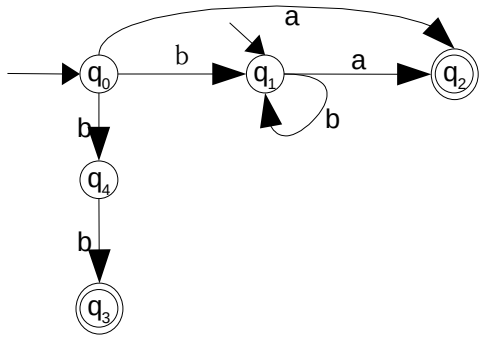
$L(N) = \{bb\} \cup \{b^n a | n \geq 0\}$

Satz:

Zu jedem ε-NEA N existiert NEA N' mit $L(N) = L(N')$

Beweisidee:

1. Übergänge in N, die mit einem Zeichen markiert sind, beibehalten.
2. Ersetze Übergänge, die mit einem Wort der Länge n markiert sind, durch $n-1$ Zustände und n Übergänge, die mit jeweils einem Zeichen markiert sind.
3. Alle Zustände, die von einem Startzustand aus durch einen oder mehrere ϵ-Übergänge erreichbar sind, werden selbst Startzustände.
4. Gelangt man von Zustand q_1 zu Zustand q_2 durch eine Kombination aus beliebig vielen ϵ-Übergängen und einem regulären Übergang, der das Zeichen a verarbeitet, so setzen wir $\delta'(q_1, a) \to q_2$.



2.3. Reguläre Ausdrücke (RA)

RA sind Formeln, mit denen Sprachen über Alphabet Σ definiert werden können.
Einem RA γ kann Sprache $L(\gamma)$ zugeordnet werden.

RA γ	$L(\gamma)$
$\emptyset$	$\emptyset$
ϵ	$\{\epsilon\}$
a	$\{a\}$
$(a\beta)$ (Konkatenation)	$\{\mu\tau \mid \mu\in L(\alpha), \tau\in L(\beta)\}$
$(\alpha+\beta)$ (Alternative)	$L(\alpha)\cup L(\beta)$
$(\alpha)^*$ (Kleene-Operator, beliebig häufige Wiederholung)	$\bigcup_{i=0}^{\infty} L(\alpha^i), \quad \alpha^i := \alpha\,\alpha^{i-1} \quad \alpha^0 = \{\epsilon\}$

Präzendenz:
* > Konkatenation > +
... erspart Klammerungen

Abkürzungen:

$(\alpha)^+ := \alpha(\alpha)^*$
$\Sigma := (t_0 + t_1 + \ldots)$ mit $t_0, t_1, \ldots$ alle Zeichen von Σ

Beispiele:

1. $\Sigma = \{0,1\}$, RA
   $0+(0+1)^*00$
   oder kurz
   $0+\Sigma^*00$
   beschreibt $\{0\}$ und mit „00“ endende Binärwörter
2. RA $(\_+a+\ldots+z+A+\ldots+Z)(\_+a+\ldots+z+A+\ldots+Z+0+\ldots+9)^*$
   legale Bezeichner in Programmiersprache

Satz: (Kleene)

Die Menge der durch reguläre Ausdrücke beschreibbaren Sprachen ist genau die Menge der regulären Sprachen REG.

Beweis per Konstruktion (siehe Literatur)

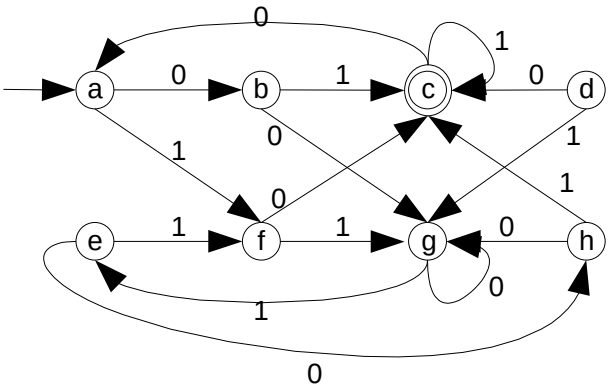
2.4. Minimierung

Satz:

Sei A ein DEA, der $L(A)$ akzeptiert. Es gibt einen eindeutigen DEA A_{min} mit minimaler Zustandsanzahl und $L(A_{min})=L(A)$.

Beweis über Konstruktion

Beispiel:



Unerreichbare Zustände weglassen: Hier: d.

Äquivalente Zustände, in Zeichen $p \equiv q$, sind solche, für die für jedes $\omega \in \Sigma^*$ gilt:

$$\delta(p,\omega) \in F \Leftrightarrow \delta(q,\omega) \in F$$

Äquivalente Zustände kann man zu einem Zustand zusammenfassen.

Konstruktionsvorschrift bildet iterativ nicht-äquivalente Zustandspaare.

Tabelle aller möglichen Paare:

	a	b	c	e	f	g
h	x		x	x	x	x
g	x	x	x	x	x	
f	x	x	x	x		
e		x	x			
c	x	x				
b	x					

Schritt 1: Paare (p,q) mit $p \in F$ und $q \notin F$ sind nicht äquivalent.

Für jedes noch nicht gekennzeichnete Paar (p,q) untersucht für jedes $a \in \Sigma$ Folgezustandspaare $(\delta(p,a),\delta(q,a)) =: (r,s)$.

Ist (r,s) nicht-äquivalent, dann auch (p,q).

Wegen (r,s) nicht-äquivalent gibt es z.B. ein ω mit $\delta(r,w) \in F$ und $\delta(s,\omega) \notin F$. Dann gilt auch $\delta(p,a\omega) \in F$ und $\delta(q,a\omega) \notin F$ und somit $p \not\equiv q$.

Schritt 2: Folgezustand betrachten

$(a,h) \xrightarrow{0} (b,g)$

$(a,h) \xrightarrow{1} (f,c)$ n.ä. → damit auch (a,h)

$(b,h) \xrightarrow{0} (g,g)$, $(b,h) \xrightarrow{1} (c,c)$

$(e,h) \xrightarrow{0} (h,g)$, $(e,h) \xrightarrow{1} (f,c)$ n.ä.

$(f,h) \xrightarrow{0} (c,g)$ n.ä.

$(g,h) \xrightarrow{0} (g,g)$, $(g,h) \xrightarrow{1} (e,c)$ n.ä.

$(a,g) \xrightarrow{0} (b,g)$, $(a,g) \xrightarrow{1} (f,e)$

$(b,g) \xrightarrow{0} (g,g)$, $(b,g) \xrightarrow{1} (c,e)$ n.ä.

usw.

Schritt 3:

$(a,g) \xrightarrow{0} (b,g)$, $(a,g) \xrightarrow{1} (f,e)$ n.ä.

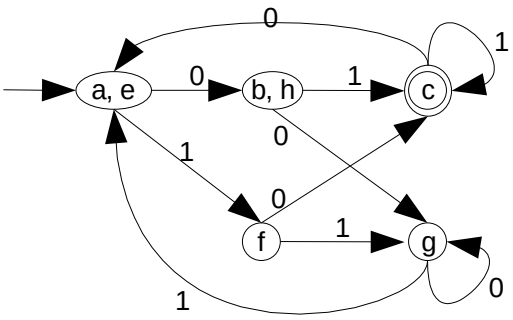
usw.

Schritt 4: gibt keine Änderung mehr

Damit folgt:

$a \equiv e, \quad b \equiv h$

Minimierter Automat:



2.5. Das Pumping-Lemma

Haupthilfsmittel, um zu zeigen, daß eine Sprache nicht regulär ist.

Satz (Pumping-Lemma für REG)

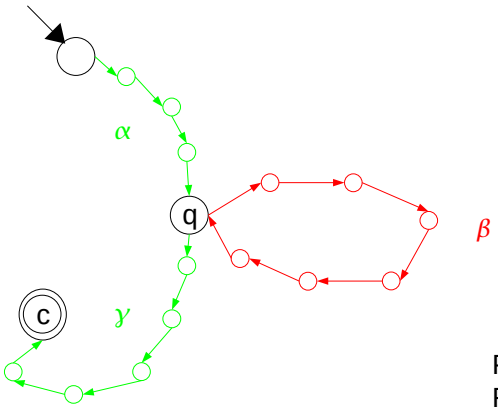
Für jede reguläre Sprache L gibt es ein $n \in N$, so daß sich alle Wörter $\omega \in L$ mit $|\omega| \geq n$ zerlegen lassen in $\omega = \alpha\beta\gamma$, so daß

1. $|\beta| \geq 1$
2. $|\alpha\beta| \leq n$
3. für alle $i = 0, 1, 2, 3, \ldots$ gilt $\alpha\beta^i\gamma \in L$.

Beweis:

Sei A ein L akzeptierender DEA mit n Zuständen. Wird Wort ω mit $|\omega| \geq n$ akzeptiert, so muß A bereits innerhalb der ersten n Zeichen ($\equiv n+1$ Zustände) eine Schleife durchlaufen.

Wir zerlegen $\omega = \alpha\beta\gamma$ so, daß der Zustand von A nach Verarbeitung von α und $\alpha\beta$ der selbe ist (q).



PUMP ⇏ REG
REG ⇒ PUMP

β (und damit die Zustandsschleife) kann beliebig oft durchlaufen werden, $\alpha\beta^i\gamma$ führt immer auf Endzustand.

Beispiel:

$L = \{0^k 1^k | k \geq 0\}$ $\quad |0^n 1^n| \geq n$

Behauptung: L ist nicht regulär.

Wir betrachten das Wort der Sprache $0^n 1^n$ mit n aus dem Pumping-Lemma. Dann müßte es bereits im ersten Teil 0^n des Wortes ein nicht-leeres β geben, das man beliebig wiederholen darf.

Offenkundig führt aber $\alpha\beta^i\gamma$ für $i \neq 1$ zu einem Wort, das nicht in L ist ($\#0 \neq \#1$). Widerspruch!

Anmerkung:

Erfüllt eine Sprache das Pumping-Lemma, so ist sie nicht notwendig regulär.

2.6. Abschlußeigenschaften

Für zwei Sprachen A, B über Σ definieren wir mit

Vereinigung: $A \cup B := \{\omega \in \Sigma^* | \omega \in A \vee \omega \in B\}$

Schnitt: $A \cap B := \{\omega \in \Sigma^* | \omega \in A \wedge \omega \in B\}$

Komplement: $\bar{A} := \Sigma^* \setminus A$

Produkt: $AB := \{\omega\gamma \in \Sigma^* | \omega \in A \wedge \gamma \in B\}$

Stern (Kleene-Hülle): $A^* := \bigcup_{n \geq 0} A^n$ mit $A^0 = \{\epsilon\}$ und $A^{n+1} = A A^n$

Definition: (Abschluss)

Eine Menge ℓ von Sprachen heißt abgeschlossen gegen eine Operation $\odot$, falls aus $A \in \ell$ und $B \in \ell$ folgt: $A \odot B \in \ell$.

(analoge Definition für unäre Operatoren)

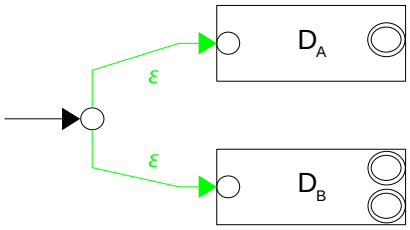
Satz:

REG (Menge der regulären Sprachen) ist abgeschlossen unter Vereinigung, Schnitt, Komplement, Produkt, Stern.

Beweis:

A, B regulär, D_A, D_B seien DEAs dafür.

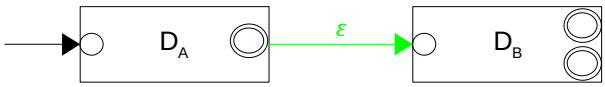
Vereinigung:



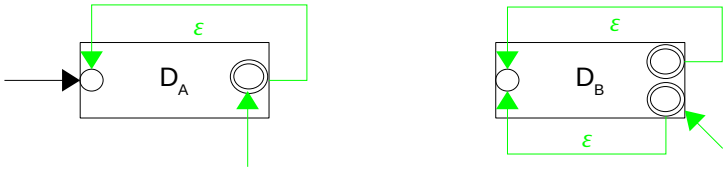
Komplement: Für $\bar{A}$ erhält man einen DEA, indem man in D_A End- und Nichtendzustände vertauscht.

Schnitt: Wegen $A \cap B = \overline{\bar{A} \cup \bar{B}}$ (de Morgan) folgt der Abschluß aus der Abgeschlossenheit für Komplement und Vereinigung.

Produkt:



Stern:



2.7. Entscheidbarkeit

Für reguläre Sprachen läßt sich das Wortproblem in linearer Zeit lösen (DEA).

Weitere entscheidbare Probleme:

- Ist eine durch ein DEA akzeptierte Sprache leer?
  (Keine Pfade von Start- zu Endzuständen)
- Ist eine reguläre Sprache endlich?
  Sie ist unendlich, falls ein Wort akzeptiert wird, das länger ist als die Anzahl der Zustände.
  Gibt es erreichbare Zyklen im DEA?
- Sind zwei reguläre Sprachen äquivalent?
  Vergleich der minimierten Automaten, oder:
  Wegen

$$\underbrace{A}_{\text{reg}} \equiv \underbrace{B}_{\text{reg}} \Leftrightarrow \underbrace{(A \cap \overline{B}) \cup (\overline{A} \cap B)}_{\text{reg}} = \emptyset$$

rückführbar auf Leerheitsproblem.

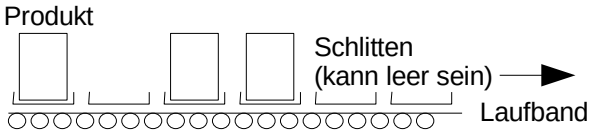
2.8. Anwendungen endlicher Automaten

2.8.1. Schaltungsentwurf

Hier: Einsatz von DEA-Varianten, die Ausgabe an Zuständen (Moore-Automaten) oder den Übergängen (Mealy-Automaten) erzeugen.

Endzustände werden nicht ausgezeichnet.

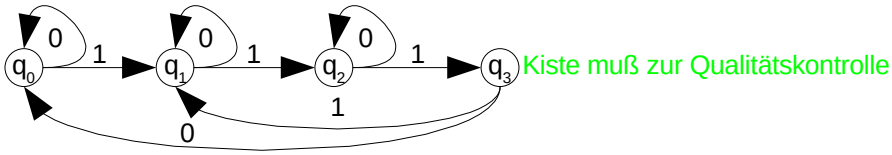
Beispiel: Produktionsanlage



Jedes 3. Produkt soll für die Qualitätskontrolle separiert werden.

Photosensor liefert { 0, Schlitten nicht belegt; 1, Schlitten belegt

Simulation der Steuerung durch DEA (verarbeitete Eingabe ist der Sensor-Input)



Binäre Codierung der Zustände: $q_0 = 00, \ldots q_3 = 11$
Input Sensor =: i; „alte“ Zustandsbits p_1, p_0 und i
definieren „neue“ Zustandsbits n_1, n_0.

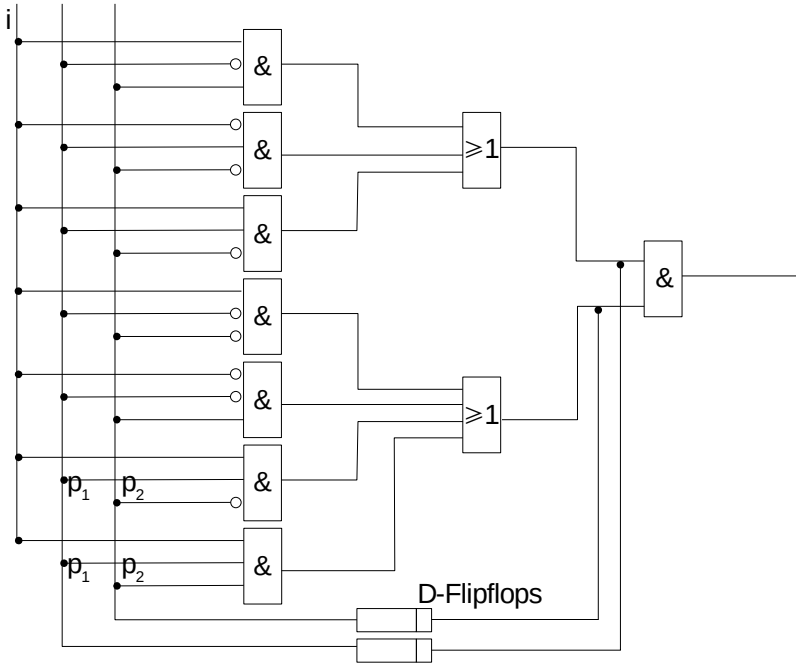
δ:

p_1	p_0	i	n_1	n_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	1	0	1

Disjunktive NF von n_1 und n_0:

$$n_1 = \overline{p_1}\, p_0\, i + p_1\, \overline{p_0}\, \overline{i} + p_1\, \overline{p_0}\, i$$

$$n_0 = \overline{p_1}\, \overline{p_0}\, i + \overline{p_1}\, p_0\, \overline{i} + p_1\, \overline{p_0}\, i + p_1\, p_0\, i$$



2.8.2. Mustererkennung

Suche nach:

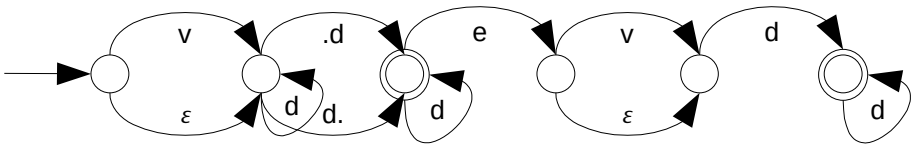
- Zeichenketten, z.B. „10100111“ oder „mama“
- regulären Ausdrücken

Beispiel:

vereinfachter RA für eine Gleitpunktzahl

$$(\epsilon + v)\,d^*(.\,d + d.)\,d^*(\epsilon + e(\epsilon + v)\,d^+)$$

- v steht für ein Vorzeichen $\{+,-\}$
- d steht für eine Ziffer $(0+1+\ldots+9)$
- Ausdruck erfaßt nicht 123e4. ein Dezimalpunkt muß vorkommen.



RA → ε-NEA → NEA → DEA läßt sich im Computer simulieren
alle Umsetzungen lassen sich mechanisieren

Vergleiche zur Textmustersuche (siehe TAD):

- Automat arbeitet sich immer in linearer Zeit durch Text ($O(n)$).
  (Beste Algorithmen für konstante Zeichenketten $O(\frac{n}{m})$).
- Automaten sind viel mächtiger (können RA erkennen).

3. Kontextfreie Sprachen

Mit einer Sprache aus REG kann man nicht die Menge aller korrekt geklammerten arithmetischen Ausdrücke erzeugen. Denn: Mit nur endlich vielen Zuständen kann man sich nicht eine beliebige Anzahl geöffneter Klammern merken.

Nächstgrößere Klasse: Kontextfreie Sprachen (CF)

Kontextfreie (Typ-2-)Grammatiken (KFG) $G=(V,\Sigma,R,S)$ haben Regeln der Form

$$A \to \alpha$$

mit $A \in V$ und $\alpha \in (V \cup \Sigma)^+$.

Ableitungen in KFG lassen sich als Syntax- oder Ableitungsbäume darstellen (s.o.)

Für einen gegebenen Syntaxbaum gibt es eindeutige

Linksableitung: „immer die am weitesten links stehende Variable zuerst ersetzen“

Rechtsableitung: „immer die am weitesten rechts stehende Variable zuerst ersetzen“

Beispiel:

C-Syntax

<stmt> → <sel_stmt> | ...

<sel_stmt> → `if (<exp>) <stmt>` | `if (<expr>) <stmt> else <stmt>`

Der Ausdruck

`if (<expr>) if (<expr>) <stmt> else <stmt>`

kann auf zwei Weisen erhalten werden:

```
<sel_stmt>
     |
if (<expr>) <stmt>
              |
          <sel_stmt>
               |
          if (<expr>) <stmt> else <stmt>
```

„`else` zum inneren `if`“

```
<sel_stmt>
     |
if (<expr>) <stmt> else <stmt>
              |
          <sel_stmt>
               |
          if (<expr>) <stmt>
```

„`else` zum äußeren `if`“

Grammatik ist (inhärent) mehrdeutig (mehrere Syntaxbäume und damit Links- und Rechtsableitungen für ein Wort möglich).

3.1. Normalformen

Man kann die Form der Regeln einschränken, ohne an Ausdrucksmächtigkeit zu verlieren.

Definition: (Chomsky-Normalform (CMF))

Eine KFG G mit $\epsilon \notin L(G)$ heißt in CNF, falls alle Regeln eine der beiden Formen haben:

$$A \to BC$$

oder

$$A \to a$$

mit $A, B, C \in V,\ a \in \Sigma$.

Ableitungsbäume sind Binärbäume. Alle Knoten (außer denen unmittelbar oberhalb der Blätter) haben 2 Kinder.

Satz:

Zu jeder KFG G mit $\epsilon \notin L(G)$ gibt es CNF-Grammatik G' mit $L(G)=L(G')$.

Beweisidee:

Nutzlose Regeln eliminieren: z.B. $A \to B$, und B kommt auf keiner linken Seite mehr vor. Allgemein $A \to \omega$, und ω enthält Variablen, die nicht in linken Seiten vorkommen.

Kettenregeln eliminieren: Für Regeln der Form $A \to B$:

1. Ersetze Variablen in Zyklen $B_1 \to B_2,\ B_2 \to B_3, \ldots, B_{k-1} \to B_k, B_k \to B_1$ in den Regeln durch eine einzige neue Variable B.
2. Nummeriere Variablen so, daß aus $A_i \to A_j$ folgt: $i < j$
3. Wir eliminieren nun von hinten nach vorne: Gilt $A_i \to A_j$ und $A_j \to \alpha$, so ersetzen wir $A_i \to A_j$ mit $A_i \to \alpha$.

Mischformen eliminieren: Wir ersetzen Regeln der Form $A \to \alpha$ (mit α kein einzelnes Terminal oder zwei Variablen) durch $A \to B_1 B_2 \ldots B_k;\ k \geq 2$

1. Für jedes $a \in \Sigma$ fügen wir neue Variable B_a zu V hinzu und die Regel $B_a \to a$.
2. Ersetze in Mischregeln $A \to \alpha$ alle a's in α durch B_a.

Mehr als zwei Variablen: Nicht in CNF sind nur noch Regeln $A \to B_1 B_2 \ldots B_k$ mit $k > 2$.

Wir führen neue Variable $C_2, \ldots, C_{k-1}$ ein und ersetzen solche Regeln durch

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

Beispiel: KFG

$S \to C$
$C \to aCb | ab$

1. Keine nutzlosen Regeln
2. Keine Zyklen, Kettenregel $S \to C$ eliminieren
   $S \to aCb | ab$
   $C \to aCb | ab$
3. Mischformen eliminieren: Führe Variablen B_a, B_b ein
   $S \to B_a C B_b | B_a B_b$
   $C \to B_A C B_b | B_a B_b$
   $B_a \to a$
   $B_b \to b$
4. Regeln mit >2 Variablen rechts ersetzen
   $S \to B_a C_1 | B_a B_b$
   $C \to B_a C_1 | B_a B_b$
   $B_a \to a$
   $B_b \to b$
   $C_1 \to C B_b$

Anmerkung:

Neben der CNF gibt es noch die Greibach-Normalform, bei der alle Regeln die Form
$A \to a B_1 \ldots B_k \quad (k \geq 0)$
haben.

3.2. Pumping-Lemma für CF

Hilfsmittel, um nachzuweisen, daß eine Sprache nicht kontextfrei ist.

Satz:

Sei L eine kontextfreie Sprache. Dann gibt es ein $n \in \mathbb{N}$, so daß sich alle Wörter $\omega \in L$ mit $|\omega| \geq n$ zerlegen lassen in $\omega = \alpha\mu\beta\nu\gamma$ mit folgenden Eigenschaften:

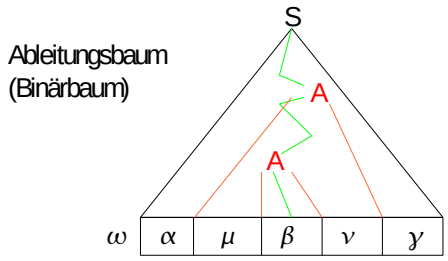
1. $|\mu\nu| \geq 1$
2. $|\mu\beta\nu| \leq n$
3. Für alle $i \geq 0$ gilt: $\alpha\mu^i\beta\nu^i\gamma \in L$

Beweisidee:

Sei G die KFG von L in CNF (Ableitungsbaum ist Binärbaum)

Sehr lange Wörter ⇒ Höhe des Baums > Anzahl Variablen in G

⇒ irgendeine Variable A kommt auf einem Pfad von der Wurzel S zu einem Blatt $\in \Sigma$ zweimal vor.



Statt

$$A \Rightarrow^* \mu A \nu \Rightarrow^* \mu \beta \nu$$

ist auch die Ableitung

$$A \Rightarrow^* \mu A \nu \Rightarrow^* \mu \mu A \nu \nu \Rightarrow^* \ldots \Rightarrow^* \mu^i A \nu^i \Rightarrow^* \mu^i \beta \nu^i$$

möglich.

Konsequenz: $\alpha \mu^i \beta \nu^i \gamma \in L$ für $i = 0, 1, 2 \ldots$

Anmerkung:

$n = 2^k$ mit k=Anzahl Variablen in G (CNF)

Beispiel:

Behauptung: $L = \{a^m b^m c^m | m \in \mathbb{N}\}$ ist nicht kontextfrei.

Sei n die Konstante aus dem Pumping-Lemma, dann ist sicher

$$|a^n b^n c^n| \geq n.$$

Dann muß Zerlegung $\alpha \mu \beta \nu \gamma$ existieren, mit

(1) $|\mu \nu| \geq 0$ und

(2) $|\mu \beta \nu| \leq n$.

Damit kann $\mu \beta \nu$ nur ein Teilwort von $a^n b^n$ oder von $b^n c^n$ sein (d.h. es kann nie a's, b's und c's enthalten).

Wegen (1) muß $\mu \neq \epsilon$ oder $\nu \neq \epsilon$.

Aus alledem folgt: Die Anzahl von a's, b's und c's in

$$\alpha \mu^2 \beta \nu^2 \gamma$$

kann nicht gleich sein.

Widerspruch! $L \notin \text{CF}$

3.3. Abschlußeigenschaften

Satz:

CF ist abgeschlossen unter

- Vereinigung
- Produkt
- Stern

CF ist nicht abgeschlossen unter

- Schnitt

- Komplement

Beweis:

Seien $G_1=(V_1,\Sigma,R_1,S_1)$ und $G_2=(V_2,\Sigma,R_2,S_2)$ zwei KFG ($V_1\cap V_2=\emptyset$).

Vereinigung: $G=(V_1\cup V_2\cup\{S\},\Sigma,R_1\cup R_2\cup\{S\to S_1|S_2\},S)$ ist KFG mit $L(G)=L(G_1)\cup L(G_2)$

Produkt: $G=(V_1\cup V_2\cup\{S\},\Sigma,R_1\cup R_2\cup\{S_1 S_2\},S)$ ist KFG mit $L(G)=L(G_1)\cdot L(G_2)$

Stern: $G=(V_1\cup\{S\},\Sigma,R_1\cup\{S\to SS|S_1|\epsilon\},S)$ ist KFG mit $L(G)=(L(G_1))^*$

Schnitt: Der Schnitt der beiden kontextfreien Sprachen

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

und

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

ist

$$\{a^j b^j c^j|j>0\}$$

und damit nicht kontextfrei (siehe oben).

Komplement: Wäre CF abgeschlossen gegen Komplementbildung, dann auch wegen dem Abschluß gegen die Vereinigung und dem Satz von de Morgan ($L_1\cap L_2=\overline{(\overline{L_1}\cup\overline{L_2})}$) gegen den Schnitt.
Widerspruch!

3.4. Der CYK-Algorithmus

Cocke, Younger, Kasam

Effizienter Algorithmus für das Wortproblem bei kontextfreien Sprachen, deren Grammatik $G=(V,\Sigma,R,S)$ in CNF gegeben ist.

Methode des dynamischen Programmierens.

Sei $\omega=a_0 a_1\ldots a_{n-1}\in\Sigma^*$. Für den Text, ob $\omega\in L(G)$ untersuchen wir, aus welchen Variablen man Teilworte $\omega_{i,j}:=a_i a_{i+1}\ldots a_{i+j-1}$ ableiten kann:

$$V_{i,j}:=\{A\in V|A\to^*\omega_{i,j}\},\quad 0\le i\le n-1\quad 1\le j\le n-i$$

Es gilt:

$$\omega\in L(G)\Leftrightarrow S\in V_{0,n}$$

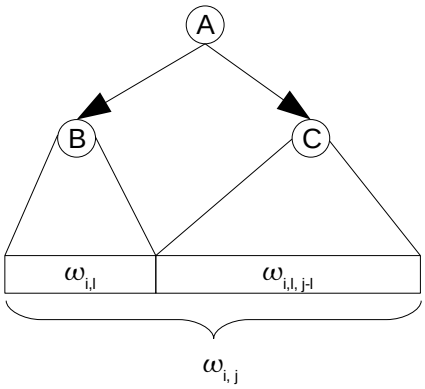
Iteratives Vorgehen in Bottom-Up-Manier:

Wegen G in CNF gilt:

$$V_{i,1}:=\{A\in V|A\to a_i\}$$

Für $j\ge 2$ und $A\in V$ gilt $A\Rightarrow^*\omega_{i,j}$ genau dann, wenn es eine Regel $A\to BC$ und Index $1\le l\le j-1$ gibt, so daß

$$B\Rightarrow^*\omega_{i,l}\qquad\text{und}\quad C\Rightarrow^*\omega_{i+l,j-l}$$



Rekursive Charakterisierung der Mengen $V_{i,j}$:

$$V_{i,j} = \bigcup_{l=1}^{j-1} \{ A \in V \mid \exists\, A \to BC \text{ mit } B \in V_{i,l} \text{ und } C \in V_{i+l,\,j-l} \}$$

Folgende Tabelle wird von oben nach unten gefüllt:

a_0	a_1	a_2	…	a_{n-2}	a_{n-1}
$V_{0,1}$	$V_{1,1}$	$V_{2,1}$	…	$V_{n-2,1}$	$V_{n-1,1}$
$V_{0,2}$	$V_{1,2}$	$V_{2,2}$		$V_{n-2,2}$	
	⋮				
$V_{0,n-2}$	$V_{1,n-2}$	$V_{2,n-2}$			
$V_{0,n-1}$	$V_{1,n-1}$				
$V_{0,n}$					

Laufzeitkomplexität: $O(n^3)$
Speicherkomplexität: $O(n^2)$

Beispiel:

Die Sprache

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

wird durch CNF-Grammatik

$$S \to AC \mid AB$$
$$C \to SB$$
$$A \to a$$
$$B \to b$$

erzeugt.

Überprüfung, ob $\omega = aabb$ in Sprache

a	a	b	b
{A}	{A}	{B}	{B}
∅	{S}	∅	
∅	{C}		
{S}			

Gibt es zwei Variablen X, Y in gleichfarbigen Menge und Regel $Z \to XY$, dann ist Z aus ■!

Wegen $S \in V_{0,4}$ gilt: $aabb$ ist aus der Sprache.

3.5. Kellerautomaten

Um Sprachen mit CF akzeptieren zu können, müssen wir endliche Automaten um einen Speicher erweitern.

Linksableitungen von CNF-Grammatiken erzeugen Wörter der Form:

$$a_1 \dots a_i A_1 \dots A_k$$

Anwendung einer Grammatikregel $A_1 \to a_{i+1}$ führt zu

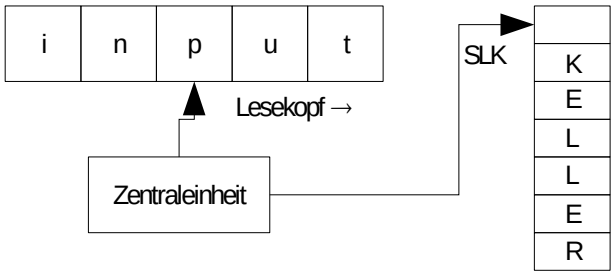
$$a_1 \dots a_i a_{i+1} A_2 \dots A_k$$

Anwendung von $A_1 \to B_1 B_2$:

$$a_1 \dots a_i B_1 B_2 A_2 \dots A_k$$

Terminalwort wird von links nach rechts aufgebaut; rechts steht eine Folge von Variablen, die nur nach links wächst oder schrumpft.

→ Erweiterung der endlichen Automaten um Stack (Kellerspeicher)!



Zustandsübergänge hängen nun nicht nur vom alten Zustand und dem Eingabezeichen ab, sondern auch vom obersten Zeichen auf dem Stack.

Definition: (Nichtdeterministischer Kellerautomat, NKA)

Ein NKA wird angegeben durch 6-Tupel: $K=(Q, \Sigma, \Gamma, \delta, q_0, \#)$ mit:

- Q: endliche Menge von Zuständen
- Σ: Eingabealphabet
- Γ: Kelleralphabet (muß nicht disjunkt zu Σ sein)
- δ: $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathrm{P}(Q \times \Gamma^*)$, die Überführungsfunktion
- q_0: Startzustand
- $\# \in \Gamma$: Kellerstartsymbol (liegt anfangs unten im Keller)

Initial enthält Keller nur #, Startzustand ist q_0, und der Lesekopf befindet sich auf dem ersten Eingabezeichen.

Es bedeutet nun

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

daß K aus dem Zustand q in q' übergehen kann, wenn Eingabezeichen a gelesen wird, und dabei das oberste Kellerzeichen A durch $B_1 \dots B_k$ ersetzt wird. (Mit B_1 oben, B_k unten.)

Unter anderem sind auch die klassischen Kelleroperationen möglich:

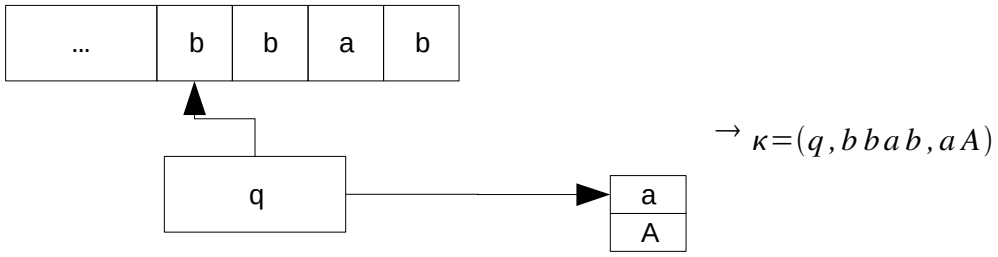
POP: Regel, bei der kein Zeichen ($=\epsilon$) auf den Keller gelegt wird

PUSH: Regel, bei der oberstes Kellerzeichen A durch BA ersetzt wird.

Eine Konfiguration $\kappa=(q,\mu,\nu)$ ist definiert durch

- q: aktueller Zustand
- μ: noch zu lesende Eingabe ($\mu\in\Sigma^*$)
- ν: Kellerinhalt ($\nu\in\Gamma^*$)

Die Anfangskonfiguration für Eingabe ω ist: $\kappa_0=(q_0,\omega,\#)$.



Übergang eines NKA von einer Konfiguration zu einer Nachfolgekonfiguration wird mit Relation $\vdash$ notiert.

Akzeptanz eines NKA kann auf zwei Arten definiert werden:

durch Endzustände: Menge $F\subseteq Q$ von Endzuständen festgelegt;
Konfigurationen $(q_F,\epsilon,\underbrace{\nu}_{\text{bel.}})$ mit $q_F\in F$ signalisieren Akzeptanz.

durch leeren Keller: Konfigurationen $(\underbrace{q}_{\text{bel.}},\epsilon,\epsilon)$ signalisieren Akzeptanz

Wie beim NEA reicht die Existenz einer akzeptierenden Berechnung, um das Wort zu akzeptieren.

Satz:

Beide Akzeptanzbedingungen für NKAs sind äquivalent. Zu jedem über leeren Keller akzeptierenden NKA gibt es einen NKA, der die selbe Sprache über Endzustände akzeptiert, und umgekehrt.

Ohne Beweis.

Beispiel:

NKA, der

$$L=\{\omega\omega^R|\omega\in\{0,1\}^*\}$$

akzeptiert, mit ω^R dem zu ω gespiegelten Wort.
NKA hat zwei Zustände:

- q^+: Zeichenweise Ablage der Eingabe im Keller (Startzustand)
- q^-: Überprüfen, ob Rest der Eingabe = Keller

Akzeptanz bei leerem Keller.

Wechsel $q^+\rightarrow q^-$ erfolgt nichtdeterministisch, wenn oberstes Kellerzeichen = Eingabezeichen.

$$K=(\{q^+,q^-\},\{0,1\},\{0,1,\#\},\delta,q^+,\#)$$

Definition von $\delta(a, b \in \{0,1\}, a \neq b)$:

$\delta(q^+, a, a) = \{(q^+, aa), (q^-, \epsilon)\}$ "Switch" n.d. Switch

$\delta(q^+, a, \#) = \{(q^+, a\#)\}$ Start

$\delta(q^+, a, b) = \{(q^+, ab)\}$ Keller füllen — ablegen

$\delta(q^-, a, a) = \{(q^-, \epsilon)\}$ Vergleich — annihilieren

$\delta(q^+, \epsilon, \#) = \{(q^+, \epsilon)\}$ Raute löschen — Ende

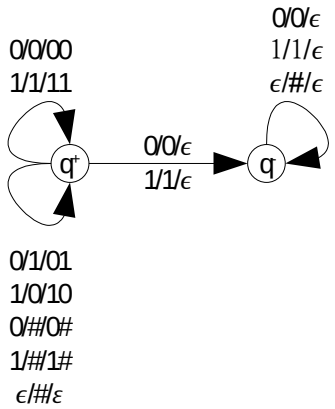
$\delta(q^-, \epsilon, \#) = \{(q^-, \epsilon)\}$ Raute löschen — Ende

$\delta(\ldots) = \emptyset$ sonst

Mögliche akzeptierende Berechnung für 0110:

$$(q^+, 0110, \#) \vdash (q^+, 110, 0\#) \vdash (q^+, 10, 10\#) \vdash (q^-, 0, 0\#) \vdash (q^-, \epsilon, \#) \vdash (q^-, \epsilon, \epsilon)$$

Anmerkung: grafische Darstellung (nicht übersichtlich, deswegen nicht wirklich verwendet)



Satz:

Eine Sprache ist kontextfrei genau dann, wenn sie von einem NKA erkannt wird.

Beweis:

„⇒“ konstruktiv

Sei KFG $G = (V, \Sigma, R, S)$ gegeben. Wir konstruieren einen NKA, der $L(G)$ akzeptiert.
Idee: Generieren Ableitung $S \Rightarrow^* \omega \in \Sigma^*$ auf dem Keller. Falls ω gleich der Eingabe, akzeptiere mit leerem Keller.

$$K = (\{q_0, q_1\}, \Sigma, V \cup \Sigma \cup \#, \delta, q_0, \#)$$

Zu δ:

1. Füge Übergang
   $$\delta(q_0, \epsilon, \#) \ni (q_1, S\#)$$
   hinzu (S auf Keller).
2. Für Regeln vom Typ $A \to \alpha$ mit $A \in V, \alpha \in (V \cup \Sigma)^*$ füge Übergang
   $$\delta(q_1, \epsilon, A) \ni (q_1, \alpha)$$
   hinzu (so können wir nichtdeterministisch alle Ableitungen auf Keller erzeugen).

3. Für alle $a \in \Sigma$ füge Übergang
   $\delta(q_1, a, a) \ni (q_1, \epsilon)$
   hinzu (Vergleich Kellerwort – Eingabe, dabei leeren des Kellers).
4. Hinzufügen von
   $\delta(q_1, \epsilon, \#) \ni (q_1, \epsilon)$

„$\Leftarrow$“: siehe Literatur

Beispiel:

Der zur Grammatik
$S \to 0S1|01$
äquivalente NKA ist
$K = (\{q_0, q_1\}, \{0,1\}, \{0,1,\#,S\}, \delta, q_0, \#)$
mit
$\delta(q_0, \epsilon, \#) = \{(q_1, S\#)\}$
$\delta(q_1, \epsilon, S) = \{(q_1, 0S1), (q_1, 01)\}$
$\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$
$\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$
$\delta(q_1, \epsilon, \#) = \{(q_1, \epsilon)\}$

Beispielberechnung für 0011:
$(q_0, 0011, \#) \vdash (q_1, 0011, S\#) \vdash (q_1, 0011, 0S1\#) \vdash (q_1, 011, S1\#) \vdash (q_1, 011, 011\#) \vdash$
$(q_1, 11, 11\#) \vdash (q_1, 1, 1\#) \vdash (q_1, \epsilon, \#) \vdash (q_1, \epsilon, \epsilon)$ → Akzeptanz

Bei Kellerautomaten:

- NKA sind echt mächtiger als deterministische Kellerautomaten (DKA).
- DKA definieren eigene Sprachklasse, die deterministisch kontextfreien Sprachen, die eine echte Teilmenge von CF sind.

Beispiel:

Für die Sprache
$L = \{\omega\omega^R | \omega \in \{0,1,\}^*\}$
läßt sich NKA konstruieren (s.o.), aber kein DKA.
Für die Sprache
$L = \{\omega\$\omega^R | \omega \in \{0,1\}^*\}$
gibt es dagegen auch einen DKA. (Umschalten vom einen in den anderen Modus nach „$“

4. Kontextsensitive und Typ-0-Sprachen

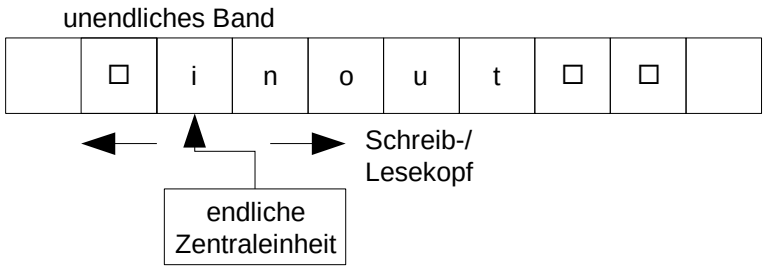
Mächtigeres Automatenmodell als Kellerautomaten nötig.

Man beseitigt Einschränkungen des KA, daß dieser nur auf oberstes Zeichen in Speicher zugreifen kann:

Der Turing-Maschine (TM) steht Speicherband mit unendlich vielen Feldern zur Verfügung. Sie kann folgende Aktionsn ausführen:

- ein Zeichen auf Band lesen, löschen, schreiben

- ein Feld nach links oder rechts rücken oder stehen bleiben.



„□“ sind Bandfelder, die noch nie vom SLK verändert wurden.

Definition:

Eine TM T ist gegeben durch ein 7-Tupel

$$T=(Q,\Sigma,\Gamma,\delta,q_0,\square,F)$$

mit

Q: endliche Zustandmenge

Σ: Eingabealphabet

$\Gamma\supset\Sigma$: Arbeitsalphabet

$\delta: Q\times\Gamma\to Q\times\Gamma\times M$ Überführungsfunktion. Dabei ist $M:=\{\leftarrow,\rightarrow,\odot\}$ die Menge der möglichen Positionsänderunge des SLK (links, rechts, stehen bleiben)

q_0: Startzustand

$\square\in\Gamma\setminus\Sigma$: Blank

$F\subseteq Q$: Menge der Endzustände

TM startet im Zustand q_0, Eingabe $\omega\in\Sigma^*$ befindet sich auf Band und der SLK steht auf erstem Zeichen von ω.

$\delta(q,a)=(q',b,m)$ bedeutet:

1. T geht von Zustand q nach q' über, wenn a auf dem Band steht.
2. Dabei wird $a\in\Gamma$ durch $b\in\Gamma$ überschrieben, und
3. der SLK führt danach Bewegung $m\in M$ aus.

Eine Konfiguration (q,γ,n) einer TM ist

- der aktuelle Zustand $q\in Q$
- der Inhalt des Bandes $\gamma\in\Gamma^*$
- und die Position des SLK n

zu einem Zeitpunkt.

(q',γ',n') ist Folgekonfiguration von (q,γ,n), falls Übergang $\delta(q,a)=(q',b,m)$ ausgeführt wird, und γ' aus γ durch Ersetzung von a mit b ans Position n entsteht. Weiter ist

$$n' = \begin{cases} n-1, & m=\leftarrow \\ n, & m=\odot \\ n+1, & m=\rightarrow \end{cases}$$

Schreibe: $(q,\gamma,n) \vdash (q',\gamma',n')$

Bequemere Schreibweise für Konfigurationen als Wörter aus $\Gamma^* Q \Gamma^*$:

$\alpha q \beta$ ist Konfiguration einer TM in Zustand q, auf dem Band steht $\alpha\beta$, und der SLK befindet sich auf dem 1. Zeichen von β.

Eine TM hält, falls in Zustand q der SLK auf Zeichen $a \in \Gamma$ steht, und δ für die Argumente (q,a) nicht definiert ist.

Definition:

Eine TM T akzeptiert ein Eingabewort $\omega \in \Sigma^*$, falls T nach endlich vielen Schritten in Endzustand hält.

Von T akzeptierte Sprache: $L(T) := \{\omega \in \Sigma^* | \omega \text{ wird von } T \text{akzeptiert}\}$

- Eine TM muß nicht halten!
- Oft sind TM so definiert, daß sie im Endzustand immer halten
- Nicht deterministische TM sind genauso mächtig wie deterministische DTM (deshalb hier keine Unterscheidung)

Beispiel:

DTM T, die $L = \{0^n 1^n 2^n | n \geq 1\}$ erkennt.

Idee: Wiederhole

- Erste 0 löschen,
- überschreibe letzte 1 mit 2
- lösche beide letzen 2en.

$0^n 1^n 2^n \rightarrow 0^{n-1} 1^{n-1} 2^{n-1}$ akzeptieren, wenn Band am Ende leer

$T = (\{q_0, \ldots, q_0, q_F\}, \{0,1,2\}, \{0,1,2,\square\}, \delta, q_0, \{q_F\})$

mit δ:

$\delta(q_0,0) = (q_1,\square,\rightarrow)$	lösche erste 0
$\delta(q_1,0) = (q_1,0,\rightarrow)$	0en überspringen
$\delta(q_1,1) = (q_2,1,\rightarrow)$	1en überspringen
$\delta(q_2,1) = (q_2,1,\rightarrow)$	
$\delta(q_2,2) = (q_3,2,\leftarrow)$	zurück zur letzten 1
$\delta(q_3,1) = (q_4,2,\rightarrow)$	letzte 1 mit 2 überschreiben
$\delta(q_4,2) = (q_4,2,\rightarrow)$	2en überspringen
$\delta(q_4,\square) = (q_5,\square,\leftarrow)$	zurück zur letzten 2
$\delta(q_5,2) = (q_6,\square,\leftarrow)$	letzten beiden 2en löschen
$\delta(q_6,2) = (q_7,\square,\leftarrow)$	⋮
$\delta(q_7,\square) = (q_F,\square,\odot)$	alle 0en, 1en, 2en gelöscht. Akzeptanz
$\delta(q_7,2) = (q_8,2,\leftarrow)$	Wechsel $q_5 \rightarrow q_6$ nötig
$\delta(q_8,2) = (q_8,2,\leftarrow)$	nach links laufen
$\delta(q_8,1) = (q_8,1,\leftarrow)$	⋮

$\delta(q_8, 0) = (q_8, 0, \leftarrow)$ ⋮

$\delta(q_8, \square) = (q_0, \square, \rightarrow)$ von vorn beginnen

Beispielrechnung:

$$q_0012 \vdash q_112 \vdash 1\,q_22 \vdash q_312 \vdash 2\,q_42 \vdash 22\,q_4 \vdash 2\,q_52 \vdash q_62 \vdash q_7\square \vdash q_F\square$$

Akzeptanz

Blanks werden nur geschrieben, wenn sie nötig sind.

Satz:

Die durch Turingmaschinen akzeptierten Sprachen sind genau die Typ-0-Sprachen.

(ohne Beweis)

Was ist mit kontextsensitiven Sprachen (Typ-1)? Da deren Grammatiken keine verkürzenden Regeln enthalten, reichen Turingmaschinen, bei denen das Band die Länge der Eingabe hat (plus linkes und rechtes Randsymbol), sogenanntes linear beschränkte Automaten (LBA).

Zusammenfassung:

Sprache	≙	Automat
regulär		endlicher Automat
kontextfrei		NKA
kontextsensitiv		LBA
Typ-0		TM

Äquivalenz deterministisch – nichtdeterministisch

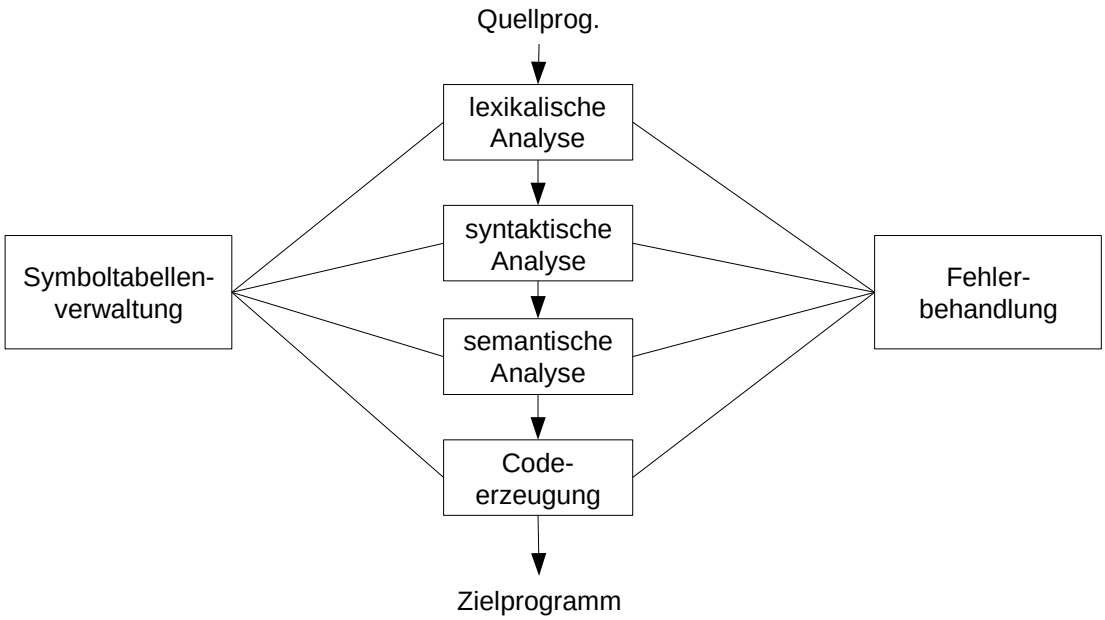
Endliche Automaten	ja
Kellerautomaten	nein
LBA	ungeklärt
Turingmaschine	ja

5. Anwendung formaler Sprachen: Compilerbau

Aho, Sethi, Ullman: Compilers
Dragon Book

Übersetzung

Quellprogramm →
ausführbares Programm



Formale Sprachen sind für die beiden ersten Übersetzungsphasen von besonderer Bedeutung.

5.1. Lexikalische Analyse

Lineare Analyse, scanning

Strom der Eingabezeichen des Quellprogramms wird linear von links nach rechts durchlaufen, und in Symbole (tokens, z.B. Schlüsselwörter, Bezeichner, Literale) aufgeteilt.

Gestalt der Tokens wird meist über reguläre Ausdrücke definiert (s.o., RA für Bezeichner und Gleitpunktzahlen) → Scanner sind DEAs

Es gibt Scanner-Generatoren (z.B. (f)lex unter UNIX), die

- reguläre Ausdrücke als Eingabe erhalten,
- intern einen NEA,
- und daraus einen DEA (in Form eines C-Programms)

erzeugen.

5.2. Syntaktische Analyse

Hierarchische Analyse, Parsing

Strom der Symbole des Scanners wird hierarchisch zu ineinander geschachtelten Gruppen zusammengefaßt (grammatische Sätze).

Syntax wird über KFG definiert.

Beispiele:

- arithmetische Ausdrücke (s.o.)
- `if-then-else` (s.o.)

Überprüfung syntaktischer Korrektheit ist daher ein Wortproblem für eine kontextfreie Sprache. Könnte man mit CYK entscheiden, aber zu teuer ($O(n^3)$).

5.2.1. Top-Down-Parser

Besser: Geschicktes implizites Erstellen des Ableitungsbaumes.

Schreibe für jede Variable eine eigene rekursive Prozedur (Methode des rekursiven Abstiegs, recursive descent).

Beispiel:

$$E \rightarrow \underbrace{\text{not}}_{\text{Tokens!}}\ E \mid (\ E\ F\ ) \mid \underbrace{\text{true}}_{\text{Tokens!}} \mid \underbrace{\text{false}}_{\text{Tokens!}}$$

$$F \rightarrow \underbrace{\text{and}}_{\text{Tokens!}}\ E \mid \underbrace{\text{or}}_{\text{Tokens!}}\ E$$

Terminale: `not, and, or, true, false, (, )`

Für jede Variable Parse-Prozedur schreiben:

```
procedure parse_E() {
     if (token == "not") {
          consume_token("not");
          parse_E();
     }
     else if (token == "(") {
          consume_token("(");
          parse_E();
          parse_F();
          consume_token(")");
     }
     else if (token == "true")
          consume_token ("true");
     else if (token == "false");
          consume_token ("false");
     else
          ... // Fehler
}

procedure parse_F() {
     if (token == "and") {
          consume_token("and");
          parse_E();
     }
     if (token == "or") {
          consume_token("or");
          parse_E();
     }
}
```

Die globale Variable `token` enthält das Lookahead-Symbol. Bestimmt, welche Ersetzung weiterverfolgt wird.

`consume_token`

- gibt Fehler aus, wenn Parameter ungleich `token`
- liest neues Token nach `token`

5.2.2. Prädiktive Parser

Existieren mehrere rechte Seiten zu einer Variable, so kann es sein, daß man zunächst eine falsche Ersetzung vornimmt, und dann den Analyseprozeß zurücknehmen muß (backtracking).

Nicht erforderlich bei prädiktiven Parsern.

Voraussetzung:

Für jedes Terminal a und jede Variable A steht zu jeder Zeit fest, welche der alternativen Ersetzungen $A \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_n$ als einzige die mit a beginnende Terminalfolge herleiten kann.

Ist bei den meisten Programmiersprachen der Fall.

Wichtiges Definitionskriterium für LL(1)-Grammatiken (Eingabe von links lesen, Linksableitung erzeugen, Lookahead ist 1 Symbol).

- Bei geeigneter Grammatikwahl hat das Parsen lineare Komplexität.
- Parser-Generatoren erzeugen für eine in (E)BNF spezifizierte KFG Parser-Programme. Beispiele: yacc / bison

5.3. Weitere Phasen und Aufgaben eines Compilers

Semantische Analyse: Typüberprüfungen

- Sind Operatoren eines Operanden zulässig?
- Ist Bezeichner von Benutzer definiert worden?
- Stimmen Formal- und Aktualparameter eines Funktionsaufrufes überein?

Diese Bedingungen können nicht durch KFG sichergestellt werden.

Symboltabellenverwaltung: Attribute der benutzten Bezeichner

- Typ
- Gültigkeitsbereich
- bei Funktionen: Signatur

Wird in allen Übersetzungsphasen genutzt.

Fehlerbehandlung: Bei Erkennen eines Fehlers:

- möglichst exakte Informationen ausgeben (Lokalisierung, Grund, Behebung)
- nicht abbrechen, weitere Fehler finden.

Code-Erzeugung: Verläuft in mehreren Schritten

(Zwischencode-Erzeugung, Optimierung, eigentliche Code-Erzeugung) → Maschinen- oder Assemblercode.

6. Abstrakte Rechner- / Programmiermodelle

6.1. Abzählbarkeit

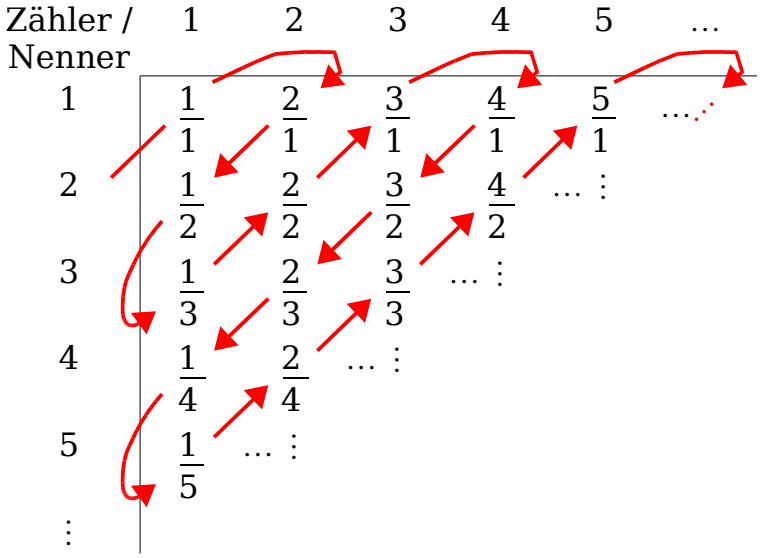
Mathematischer Begriff, der im folgenden öfter gebraucht wird.

Definition:

Eine Menge M heißt abzählbar, wenn es eine surjektive Abbildung $f:\mathbb{N}\to M$ gibt.

Beispiele:

1. endliche Menge
2. $\mathbb{N}$
3. Menge der (positiven) rationalen Zahlen ist abzählbar!
   Nummerierungsschema



Nummerierungsreihenfolge

„Diagonalisierungstrick“

jeder Zahl aus $\mathbb{Q}^+$ kann eine Zahl aus $\mathbb{N}$ zugeordnet werden

4. Ist $\mathbb{R}$ abzählbar? Nein, denn dies gilt nicht einmal für die reellen Zahlen in $[0,1]$. Denn sonst gäbe es eine surjektive Funktion $f:\mathbb{N}\to[0,1]$, d.h. $\{f(1),f(2),f(3),\ldots\}=[0,1]$

   Schreibe Binärdarstellungen von $f(1),f(2),f(3),\ldots$ untereinander (hier nur Nachkommastellen):

f(1):	0	1	1	0	0	0	…..
f(2):	1	1	0	1	0	0	…..
f(3):	0	0	1	0	0	0	…..
f(4):	1	0	1	1	0	1	…..
…..	…..	…..	…..	…..	…..	…..	…..

Bilde Komplementärzahl der Diagonale (im Beispiel: 1 0 0 0 .....).
Diese Zahl ist zu keinem $f(i)$ identisch → Widerspruch!

$\mathbb{R}$ ist über-abzählbar.

Anmerkung:

In der theoretischen Informatik ist $\mathbb{N}=\mathbb{N}_0$.

6.2. Berechenbarkeitsbegriff

Beschränkung auf Funktionen $f:\mathbb{N}^k\to\mathbb{N}$. Alle Berechnungsprobleme können darauf zurückgeführt werden.

Ist eine Funktion $f:\mathbb{N}^k\to\mathbb{N}$ für jedes Eingabe-Tupel definiert, so heißt sie total, sonst partiell.

Für ein Tupel $(n_1,\ldots,n_k)$, für das f nicht definiert ist, schreiben wir
$f(n_1,\ldots,n_k)=\perp$

Definition:

Eine partielle Funktion $f:\mathbb{N}^k\to\mathbb{N}$ heißt berechenbar, falls ein Algorithmus (Programm) existiert, der für Eingabe $(n_1,\ldots,n_k)$

- nach endlich vielen Schritten mit der Ausgabe $f(n_1,\ldots,n_k)$ hält, falls $f(n_1,\ldots,n_k)\neq\perp$
- endlos läuft oder ohne Ausgabe hält, falls $f(n_1,\ldots,n_k)=\perp$.

Beispiele:

1. Die Ackermannfunktion $\text{ack}:\mathbb{N}\times\mathbb{N}\to\mathbb{N}$
   $\text{ack}(0,m)=m+1$
   $\text{ack}(n,0)=\text{ack}(n-1,1)$ falls $n\geq 1$
   $\text{ack}(n,m)=\text{ack}(n-1,\text{ack}(n,m-1))$, falls $n,m\geq 1$
   ist berechenbar.
2. Die überall undefinierte Funktion
   $f_\perp:\mathbb{N}\to\mathbb{N}$, $f_\perp(n)=\perp$ für alle $n\in\mathbb{N}$
   ist berechenbar. (Z.B. durch Programm, das für keine Eingabe terminiert.)
3. Die $(3n+1)$-Funktion ist berechenbar, aber man weiß nicht, ob Berechnung für jedes n hält.

```
int achterbahn (int n) {
     if (n == 1)
          return 1;
     if (n % 2 == 0)
          return achterbahn(n/2);
     else
          return achterbahn(3*n+1);
}
```

4. Es muß Funktionen geben, die nicht berechenbar sind.

Betrachte Funktionenschar $f_r: \mathbb{N} \to \mathbb{N}$ mit

$$f_r(n) = \begin{cases} 1, & \text{falls } n \text{ Anfangsstück des Nachkommateils von } r \in \mathbb{R} \text{ ist,} \\ 0, & \text{sonst.} \end{cases}$$

(z.B. $f_\pi(1)=1$, $f_\pi(14159)=1$, $f_\pi(26)=0$)

Menge der paarweise verschiedenen Funktionen

$$\{f_r \mid r \in [0,1]\}$$

ist überabzählbar; Menge aller Programme, die man in endliches Alphabet niederschreiben kann, dagegen nur abzählbar.

6.3. Turingmaschinen

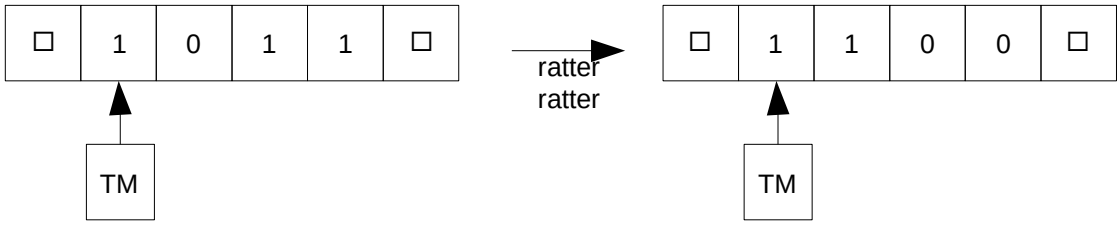
6.3.1. Turing-Berechenbarkeit

Bisher: Turing-Maschinen akzeptieren Wörter (Typ-0-Sprachen)

Nun: TM berechnen Funktionen

Beispiel:

DTM, die zu einer Binärzahl 1 addiert



- Gehe zum rechten Ende (q_0)
- Nach links laufen, und 1en in 0en umwandeln, bis man auf die erste 0 oder □ stößt (q_1)
- 0 / □ in 1 umwandeln (q_2)
- SLK auf erstes Zeichen setzen (q_3)

$T = (\{q_0, \ldots, q_3\}, \{0,1\}, \{0,1,\square\}, \delta, q_0, \{q_3\})$

$\delta(q_0, 0) = (q_0, 0, \rightarrow)$ zum rechten Ende laufen
$\delta(q_0, 1) = (q_0, 1, \rightarrow)$ zum rechten Ende laufen
$\delta(q_0, \square) = (q_1, \square, \leftarrow)$ Addiermodus starten
$\delta(q_1, 1) = (q_1, 0, \leftarrow)$ 1en in 0en umwandeln
$\delta(q_1, 0) = (q_2, 1, \leftarrow)$ erste auftretende 0 in 1 umwandeln
$\delta(q_1, \square) = (q_3, 1, \odot)$ SLK richtig positioniert
$\delta(q_2, 0) = (q_2, 0, \leftarrow)$ zum Anfang laufen
$\delta(q_2, 1) = (q_2, 1, \leftarrow)$ zum Anfang laufen
$\delta(q_2, \square) = (q_3, \square, \rightarrow)$ SLK richtig positioniert

Ausgabe steht rechts vom SLK

Eine TM $T=(Q,\Sigma,\Gamma,\delta,\square,q_0,F)$ definiert eine partielle Funktion $f_T:\Sigma^*\to(\Gamma\setminus\{\square\})^*$, also eine Abbildung von einem Ein- auf ein Ausgabewort. Das Ausgabewort wird als das längste Wort über $\Gamma\setminus\{\square\}$ definiert, das nach Terminierung rechts vom SLK steht (inklusive dem Zeichen unter dem SLK). Ist für ein $\omega\in\Sigma^*$ die Berechnung verwerfend (Halten im Nicht-Endzustand) oder unendlich, so schreiben wir

$$f_T(\omega)=\perp$$

Umgekehrt definieren wir:

Definition:

Seien Σ und Λ zwei Alphabete und

$$f:\Sigma^*\to\Lambda^*$$

eine partielle Funktion f. Sie heißt Turing-berechenbar, falls es DTM T mit Eingabealphabet Σ, einem Arbeitsalphabet Γ und Blanksymbol $\square$, so daß $\Lambda\subseteq\Gamma\setminus\{\square\}$ und $f=f_T$.

Beispiel für eine nicht Turing-berechenbare Funktion:

Die Busy Beaver-Funktion $\mathrm{bb}:\mathbb{N}\to\mathbb{N}$.

$\mathrm{bb}(n)$ gibt an, wie viele 0en eine DTM mit n Zuständen und Arbeitsalphabet $\{0,\square\}$ maximal auf ein anfangs leeres Band schreiben kann. Es wird vorausgesetzt, daß die DTM irgendwann hält.

$\mathrm{bb}(1)=1$
$\mathrm{bb}(2)=4$
$\vdots$

Warum ist die bb nicht Turing-berechenbar?

6.3.2. Programmierung mit TM

Konzepte höherer Programmiersprachen können in TMen ausgedrückt werden.

6.3.2.1. Hintereinanderschaltung

Seien T_1, T_2 zwei TMen, $T_i=(Q_1,\Sigma,\Gamma,\delta_i,q_{0,i},\square,F_i)$. Eine TM, die sich zunächst wie T_1, dann wie T_2 verhält und die partielle Funktion

$$f_{T_2}\circ f_{T_1}$$

berechnet, ist

$$T_1;T_2:=(Q_1\cup Q_2,\Sigma,\Gamma,\delta,q_{0,1},\square,F_2)$$

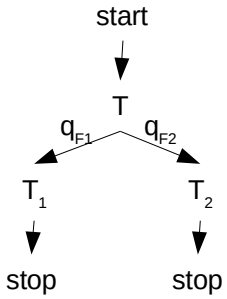
mit

$$\delta(q,a)=\begin{cases}\delta_1(q,a), & \text{falls } q\in Q_1\setminus F_1\\ \delta_2(q,a), & \text{falls } q\in Q_2\\ (q_{0,2},a,\odot), & \text{falls } q\in F_2\end{cases}$$

Durch dreimaliges Hintereinanderschalten der TM, die 1 zu einer Binärzahl addiert, erhält man eine TM, die 3 addiert.

6.3.2.2. Bedingte Anweisungen

Analog kann zusammengesetzte TM



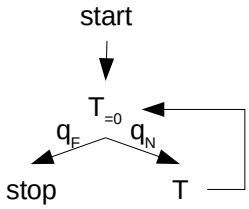
konstruiert werden, die je nach eingenommenem Endzustand von T wahlweise T_1 oder T_2 ausführt.

6.3.2.3. Schleifen

Mit folgender TM $T_{=0}$, die ein Band überprüft, ob nur eine 0 darauf steht:

$$\delta(q_0,a)=(\overbrace{q_N}^{\text{nicht akz.}},a,\odot) \text{ für } a\neq 0$$
$$\delta(q_0,0)=(q_1,0,\rightarrow)$$
$$\delta(q_1,a)=(q_N,a,\odot) \text{ für } a\neq\square$$
$$\delta(q_1,\square)=(\underbrace{q_F}_{\text{Endzust.}},\square,\odot)$$

Kann mit



eine TM „WHILE Band ≠ 0 DO T“ konstruiert werden.

Varietät: Mehrband-TMen mit k Bändern und k unabhängig positionierbaren SLKen.

Jede Mehrband-TM kann durch Einband-TM simuliert werden.

6.4. Registermaschinen (RM)

Cook, Reckhow (1973)

Auch Random Access Machines (RAM) genannt.

Einfaches Rechnermodell, an realen Rechnern orientiert.

6.4.1. Definition

Die RM hat

- unendlich viele Registerzellen $r_1, r_2, \ldots$
- einen Akkumulator r_0
- einen Satz elementarer Befehle und
- einen Programmzähler b.

Inhalt von r_i wird mit $c(i)$ bezeichnet („content“) und kann beliebigen Wert $\in \mathbb{N}$ annehmen.

Es wird festgelegt,

- welche Registerzellen welche Eingabewerte enthalten (die anderen sind initial 0)
- welche Registerzelle nach Programmablauf die Ausgabe enthält.

RM kann ein Programm ausführen

- besteht aus elementaren Befehlen
- Zeilen nummeriert

Befehlszähler gibt an, welche Programmzeile als nächstes bearbeitet wird. (initial: 1)

Operanden eines Befehls können gegeben sein durch:

Operandenart	geschrieben	„eigentlicher“ Wert
Konstante	#k	k
direkte Adressierung	i	c(i) = Wert von Register i
indirekte Adressierung	*i	c(c(i)) = Wert von Register c(i)

Mit $v(x)$ („value“) bezeichnen wir den eigentlichen Wert des Arguments x.

Befehl	Effekt
LOAD x	`c(0) = v(x); b++;`

STORE i
STORE *i
	`c(i) = c(0); b++;`
	`if (c(i) > 0) {`
	`  c(c(i)) = c(0);`
	`  b++;`
	`}`
	`else`
	`  b = ∞;  //Programmende`

ADD x	`c(0) += v(x); b++;`
SUB x	`c(0) = max {0, c(0) – v(x)};` `b++;`
MULT x	`c(0) *= v(x); b++;`

x: beliebige Operandenart

i: direkte Adresse

*i: indirekte Adresse

l: Zeilennummer

DIV x	`if (v(x) > 0) {`
	`  c(0) /= v(0);`
	`  b++;`
	`}`
	`else`
	`  b = ∞;`

GOTO l
JZERO l
(jump if zero)
	`b = l;`
	`if (c(0) == 0)`
	`  b = l;`
	`else`
	`  b++;`

END	`b = ∞;`

Beispiele:

1. $f(\underbrace{m}_{\text{initial i } r_1}, \underbrace{n}_{r_2}) := \underbrace{m+n+1}_{r_3}$

1	LOAD	1
2	ADD	2
3	STORE	1
4	LOAD	#1
5	ADD	1
6	STORE	3

2. Zuweisung:
   `c(i) := c(j) * c(k)`

⋮		
l	LOAD	j
l+1	MULT	k
l+2	STORE	i

3. Schleife:
   `while (c(i) > c(j)) { ... }`

LOOP:	LOAD	i
	SUB	j
	JZERO	END_LOOP
	⋮	
	⋮	Schleifen-Body
	⋮	
	GOTO	LOOP
END_LOOP:		

RM_Programme ähneln Einadress-Assemblerprogrammen.
Hochsprachenprogramme können in RM-Programme übersetzt werden und umgekehrt (d.h. gleiche Mächtigkeit).

6.4.2. RM-Berechenbarkeit

Eingabe $n_1,\ldots,n_k$ stehe anfangs in $r_1,\ldots,r_k$; Ausgabe nach Terminierung in

festgelegter Registerzelle i. kurz $c[n_1,\ldots,n_k]$

Definition:

Die durch eine RM R berechnete partielle Funktion
$f_r: \mathbb{N}^k \to \mathbb{N}$
ist gegeben durch:

- $f_R(n_1,\ldots,n_k)=\perp$, falls R für $c[n_1,\ldots,n_k]$ nicht terminiert
- $f_r(n_1,\ldots,n_k)=c(i)$, sonst

Definition:

Eine partielle Funktion $f: \mathbb{N}^k \to \mathbb{N}$ wird RM-berechenbar genannt, wenn es eine RM R gibt, mit $f_R=f$.

6.5. Äquivalenz der Berechenbarkeitsbegriffe

Konzept der RM-Berechenbarkeit ist vertrauter als Turing-Berechenbarkeit. Es gilt:

Satz:

Eine partielle Funktion $f: \mathbb{N}^k \to \mathbb{N}$ ist Turing-berechenbar $\Leftrightarrow$ f ist RM-berechenbar.

Beweis:

für die Grundidee siehe Asteroth, Baier

Simulation von RMen durch TMen und umgekehrt.

Churchsche These
Die durch formale Definition der Turing-Berechenbarkeit (äquivalent: RM-Berechenbarkeit) erfaßte Klasse von berechenbaren partiellen Funktionen stimmt überein mit der Klasse der „intuitiv“ berechenbaren Funktionen.

Anmerkung:

Weitere gleich mächtige Berechenbarkeitsbegriffe:

WHILE-, GOTO-Berechenbarkeit, µ-Rekursivität

7. Entscheidungsprobleme

7.1. (Semi-)Entscheidbarkeit

Probleme, bei denen die korrekte Antwort „Ja“ oder „Nein“ ist, heißen Entscheidungsprobleme.

Z.B. Wortproblem: Ist ein gegebenes Wort $\omega \in \Sigma^*$ in einer formalen Sprache $L \subset \Sigma^*$ enthalten?

Zusammenhang Entscheidbarkeit ↔ Berechenbarkeit
Definition:

Eine Sprache $L \subseteq \Sigma^*$ heißt entscheidbar, falls die charakteristische Funktion von $L, \chi_L : \Sigma^* \to \{0,1\}$ berechenbar ist. Dabei ist für $\omega \in \Sigma^*$:

$$\chi_L(\omega) := \begin{cases} 1, & \omega \in L \\ 0, & \omega \notin L \end{cases}.$$

Eine Sprache heißt semi-entscheidbar, falls die „halbe“ charakteristische Funktion von $L, \chi_L' : \Sigma^* \to \{1, \perp\}$, berechenbar ist. Dabei ist für $\omega \in \Sigma^*$:

$$\chi_L'(\omega) = \begin{cases} 1, & \omega \in L \\ \perp, & \omega \notin L \end{cases}.$$

Beispiele:

1. Endliche Mengen sind immer entscheidbar.
2. $L = \{n \in \mathbb{N} | n \text{ ist prim}\}$ ist entscheidbar.
3. $L = \{n \in \mathbb{N} | \text{achterbahn}(n) = 1\}$ ist (nach heutigem Kenntnisstand) semi-entscheidbar.

Für entscheidbare Sprachen muß es terminierenden Algorithmus geben (Entscheidungsverfahren), der für jedes $\omega \in \Sigma^*$ feststellt, ob $\omega \in L$ oder $\omega \notin L$.

Für semi-entscheidbare Sprachen bricht Entscheidungsverfahren nur ab, falls $\omega \in L$.

Satz:

Sprache L entscheidbar $\Leftrightarrow L$ und $\overline{L}$ semi-entscheidbar.

„⇒“ klar

„⇐“ Entscheidungsverfahren für L und $\overline{L}$ parallel laufen lassen, einer von beiden muß nach endlicher Zeit anhalten.

7.2. Halteproblem

Definiert eine wichtige entscheidbare Sprache.

Kann man ein Programm P_H schreiben, das für andere Programme feststellt, ob diese für eine bestimmte Eingabe terminieren?

Halteproblem-Tabelle (HT):

Menge aller Programme ist abzählbar: $P_0, P_1, P_2, \ldots$
Menge der endlichen Eingaben ist auch abzählbar: $I_0, I_1, I_2, \ldots$

In der unendlichen HT tragen wir an Position (i, j) ein „J“ ein, falls Programm P_i für Eingabe I_j terminiert, sonst „N“.

	⋮	⋮	⋮	⋮	⋮	⋮	⋰
	P_4	J	J	J	N	J	…
	P_3	N	J	J	J	J	…
Programm	P_2	J	N	N	J	J	…
	P_1	J	J	J	J	J	…
	P_0	J	N	N	N	J	…
		I_0	I_1	I_2	I_3	I_4	…

Mit P_H ließe sich Programm P_K schreiben, das sich <u>komplementär</u> zur Diagonale der HT verhält.

Steht an Position (i,i) ein

- „J“, so lassen wir P_K für Eingabe I_i endlos laufen
- „N“, so hält P_K für I_i an.

Für P_K selbst muß es in der HT eine Zeile l geben:
$P_K=P_l$.

Steht in (l,l) ein „J“, so müßte P_K für Eingabe I_l terminieren. Gleichzeitig (nach Komplementärkonstruktion) müßte P_H für (l,l) ein „N“ berechnen, d.h. $P_l=P_K$ endlos laufen → Widerspruch!

Steht in der HT bei (l,l) ein „N“ … → Widerspruch!

Ein Programm wie P_H ist unmöglich!

Formaler Beweis dieses <u>Halteproblems</u> über Turingmaschinen (als allgemeinstes Berechnungsmodell)

Dazu benötigt man eine Abbildung, die einer TM T eine Codierung $\omega_T \in \Sigma^*$ zuordnet: $\text{code}(T)=\omega_T$, sowie Umkehrung: $\text{decode}(\omega)=T_\omega$

<u>Definition</u> (mit TM):

Das <u>spezielle Halteproblem</u> ist die Sprache
$\mathcal{H} := \{\omega \in \Sigma^* | \text{decode}(\omega)=T_\omega \text{ angesetzt auf } \omega \text{ hält}\}$.

<u>Satz:</u>

$\mathcal{H}_s$ ist nicht entscheidbar (semi-entscheidbar).

Mit speziellem Halteproblem kann das allgemeine Halteproblem bewiesen werden:

<u>Definition:</u>

Das <u>allgemeine Halteproblem</u> ist die Sprache
$\mathcal{H} = \{\omega \# \gamma | \omega, \gamma \in \Sigma^*, \text{decode}(\omega)=T_\omega \text{ angesetzt auf Eingabe } \gamma \text{ hält}\}$.

Satz:

Auch $\mathcal{H}$ ist nicht entscheidbar.

Weitere Konsequenz: Satz von Rice

Verifikationsprobleme sind meistens unentscheidbar, wenn man keine Einschränkungen an Programme vorgibt, z.B.

- Programm berechnet konstante Funktion
- Programm hält für leere / bestimmte Eingabe
- …
- Programm „arbeitet richtig“

7.3. Postsches Korrespondenzproblem

Weiteres unentscheidbares Problem.

Gegeben: Endliche Folge von Wortpaaren $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ mit $x_i, y_i \in \Sigma^*$.

Gesucht: Folge von Indizes $i_1, i_2, \ldots i_n \in \{1, 2, \ldots, k\}$ mit $x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n}$

Beispiel:

Korrespondenzproblem mit $\left(\overset{x_1}{(1}, \overset{y_1}{101}), \overset{x_2}{(10}, \overset{y_2}{00}), \overset{x_3}{(011}, \overset{y_3}{11}) \right)$

Mögliche Lösung $(1,3,2,3)$ denn

$$x_1 x_3 x_2 x_3 = \overbrace{1}^{x_1} \overbrace{011}^{x_3} \overbrace{10}^{x_2} \overbrace{011}^{x_3} = \underbrace{101}_{y_1} \underbrace{11}_{y_3} \underbrace{00}_{y_2} \underbrace{11}_{y_3} = y_1 y_3 y_2 y_3$$

I.A. sehr komplex.

Lösung für $((001,0),(01,011),(01,101),(10,001))$ erfordert 66 Indizes!

Satz:

Postsches Korrespondenzproblem ist nicht entscheidbar.

Beweis über Unentscheidbarkeit des Halteproblems.

PCP wird häufig zum Beweis der Unentscheidbarkeit von Problemen bei formalen Sprachen verwendet.

Z.B. kann mit dem PCP gezeigt werden, daß das Schnittproblem für kontextfreie Sprachen (d.i. die Fragestellung, ob der Schnitt zweier kontextfreier Sprachen leer ist oder nicht) nicht entscheidbar ist.

8. Komplexitätstheorie

Klassifikation von Algorithmen gemäß ihres Bedarfs an „Berechnungsressourcen“ (Rechenzeit, Speicherplatzbedarf)

siehe TAD

8.1. Komplexitätsklassen und P-NP-Problem

Einteilung entscheidbarer Sprachen in Klassen.

Definition:

Sei $f: \mathbb{N} \to \mathbb{N}$ eine Funktion. Die Klasse $\mathrm{DTIME}(f(n))$ besteht aus allen Sprachen ($\hat{=}$ Berechnungsproblemen) A, für die es eine deterministische TM (DTM) T gibt mit $A = L(T)$ und $\mathrm{dtime}_T(\omega) \leq f(|\omega|)$.

Hierbei ist $\mathrm{dtime}_T: \Sigma^* \to \mathbb{N}$ die Anzahl der Rechenschritte, die T für Eingabewörter $\in \Sigma^*$ benötigt.

Definition:

Die Komplexitätsklasse P ist

$$\mathrm{P} := \{A \subseteq \Sigma^* | \text{es gibt eine DTM } T \text{ mit } L(T) = A \text{ und ein Polynom } p \text{ mit } \mathrm{dtime}_T(\omega) \leq p(|\omega|)\}$$
$$= \bigcup_{p \text{Polynom}} \mathrm{DTIME}(p(n))$$

Man sagt: P ist die Klasse der Probleme, für die effiziente Algorithmen existieren.

Ausdehnung der Definition auf nichtdeterministische TM (NTM):

Definition:

Für eine NTM T sei

$$\mathrm{ntime}_T(\omega) = \begin{cases} \min\{\text{Länge aller akzeptierenden Berechnungen von } T \text{ für } \omega\}, & \omega \in L(T) \\ 0, & \omega \notin L(T) \end{cases}$$

$\mathrm{NTIME}(f(n))$ besteht aus allen Sprachen A, für die es NTM T gibt mit $L(T) = A$ und $\mathrm{ntime}_T(\omega) \leq f(|\omega|)$.
Es ist

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

Anmerkung:

Problem aus NP können von einer DTM in $\mathrm{DTIME}(2^{p(n)})$ gelöst werden (p Polynom), also in exponentieller Zeit.

Definition: (P-NP Problem)

Unbekannt ist, ob $\mathrm{P} \subsetneq \mathrm{NP}$ oder $\mathrm{P} = \mathrm{NP}$.

Momentan größtes ungelöstes Problem der theoretischen Informatik.

Es wird angenommen, daß $\mathrm{P} \subsetneq \mathrm{NP}$.

Wichtige Probleme in NP: Travelling Salesman, Erfüllbarkeitsproblem, Reihenfolgeprobleme (s.u.)
... bisher sind dafür keine polynominalen Algorithmen bekannt.

8.2. NP-Vollständigkeit

Definition:

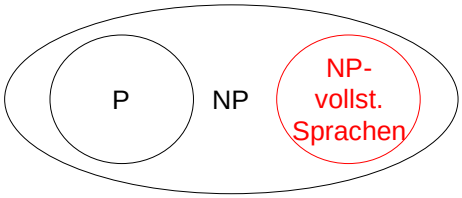
Eine Sprache A heißt NP-hart, falls das Wortproblem dafür mindestens so

aufwendig ist wie für alle Sprachen aus NP.

Eine Sprache A heißt NP-vollständig, falls A NP-hart ist und $A \in NP$ gilt.

Satz:

Sei A NP-vollständig. Dann gilt
$A \in P \Leftrightarrow P = NP$.



Beispiel für eine ganz zentrale NP-vollständige Sprache:

Definition:

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT):

- gegeben: eine Formel F der Aussagenlogik
- gesucht: eine Belegung der in F enthaltenen Variablen, so daß F wahr ist. Dann ist F erfüllbar.

Die Sprache SAT ist die Menge der erfüllbaren Formeln.

Beispiele:

$x_1 \wedge \overline{x_1} \notin SAT$
$x_1 \vee \overline{x_1} \in SAT$
$x_1 \vee (x_2 \wedge \overline{x_3}) \in SAT$

1	3	*
2	*	*

Richard Kaye

Satz (Look)

SAT ist NP-vollständig.

Es gibt eine NTM T, die SAT in polynominaler Rechenzeit erkennt, also $SAT \in NP$. Dazu rät sie eine Variablenbelegung, und verifiziert die Wahrheit des Ausdrucks (guess and check).

NP-Härte von SAT: Details siehe Schöning

8.3. Weitere NP-vollständige Probleme

Für folgende Probleme kann gezeigt werden, daß sie

- aus NP sind, und
- daß man SAT auf sie zurückführen kann,

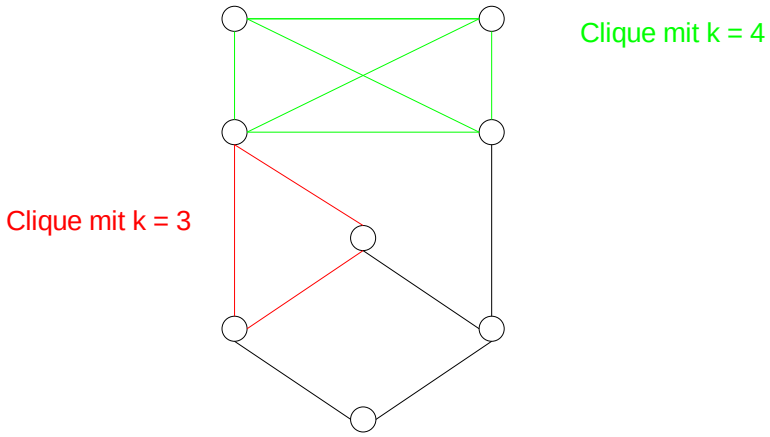
sie also auch NP-vollständig sind:

3SAT: Spezialfall von SAT

- gegeben: Aussagenlogische Formel F in konjunktiver Norm (Klauselform) mit höchstens 3 Literalen pro Klausel.
- gesucht: Variablenbelegung, die F wahr macht.

CLIQUE:

- gegeben: ungerichteter Graph $G=(V,E)$ und Zahl $k\in\mathbb{N}$
- gesucht: Besitzt G eine Clique der Größe k? (Teilmenge V' der Knoten mit $|V'|=k$ und $\forall u,v\in V'$ mit $u\neq v$ gilt: $(u,v)\in E$)



0-1-RUCKSACK: Ein Dieb hat die Wahl zwischen mehreren Gegenständen mit gegebenen Werten und Gewichten. Kann er seinen Rucksack optimal füllen?

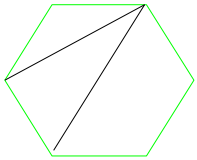
- gegeben: Zahlen $v_0,v_1,\ldots,v_k\in\mathbb{N}$ Werte
  $w_0,w_1,\ldots w_k\in\mathbb{N}$ Gewichte
  der Gegenstände
  W=Kapazität Rucksack
- gesucht: Gibt es optimale Teilmenge $I\subseteq\{0,1,\ldots,k\}$ mit $\sum_{i\in T} w_i\leq W$ und $\sum_{i\in T} v_i$ maximal?

PARTITION:

- gegeben: $a_0,a_1,\ldots,a_k\in\mathbb{N}$
- gesucht: Gibt es Teilmenge $j\subseteq\{0,1,\ldots,k\}$ mit $\sum_{i\in J} a_i=\sum_{j\notin J} a_j$

(Un-)Gerichteter Hamilton-Kreis:

- gegeben: Ein (un-)gerichteter Graph $G=(V,E)$
- gesucht: Besitzt G einen Hamilton-Kreis? (Zyklischer Pfad, der alle Knoten genau einmal enthält.



Travelling Salesman:

- gegeben: Eine $n \times m$-Matrix $(M_{i,j})$ von „Entfernungen“ zwischen n „Städten“ und eine Zahl k
- gefragt: Gibt es eine Permutation π (eine „Rundreise“), so daß
$$\sum_{i=1}^{n-1} M_{\pi(i),\pi(i+1)} + M_{\pi(n),\pi(1)} \leq k?$$

~ 1000 NP-vollständige Probleme

Es gibt entscheidbare Sprachen, die nicht in DTIME$(f(n))$ für beliebige $f(n)$ liegen. (keine obere Schranke!)

Übungen

Aufgabe 1.2

a) $S \to a$
$S \to aS$
$S \to aSS$
Typ 2: Links steht nur eine Variable

b) $S \to a$
$S \to aSS$
$\underbrace{a}_{\text{kontextsensitiv}} \quad Saa \to aaSaaa$
Typ 1: Dritte Regel hat links Terminale und Variablen

c) $S \to a$
$S \to aS$
$aaS \to SS$
Typ 0: Dritte Regel nicht kontextsensitiv, verkürzend

d) $S \to a$
$S \to aS$
Typ 3: Rechtslinear

Aufgabe 1.3

$S \to 0 \mid 1A \quad \to$ kurz für $S \to 0$
$A \to 1B \mid 0C \mid 1 \quad S \to 1A$
$B \to aA \mid 0B \mid 0$
$C \to 1C \mid 0A$

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

$0, 9, 3, 6, 15$

Die von der Grammatik beschriebene Sprache enthält alle Binärzahlen, die Vielfache von 3 sind.

$\underbrace{100101\ldots101}_{=b} V \quad \leftarrow$ Variable am Ende $\quad V \in \{S, A, B, C\}$

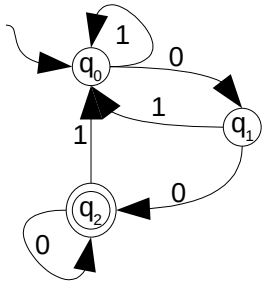
Interpretation:
S: erzeugt 0 oder eine mit 1 beginnende Binärzahl
A: $b \bmod 3 = 1$
B: $b \bmod 3 = 0$
C: $b \bmod 3 = 2$

Beachten Sie die Konsistenz: A signalisiert $b \bmod 3 = 1$

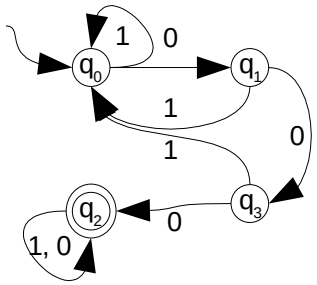
- Durch Anhängen einer 1 hat man $(b \cdot 1) \bmod 3 = (2b+1) \bmod 3 = (2(b \bmod 3)+1) \bmod 3 = 0$. Daher setzt man Schlußvariable auf B oder bricht die Ableitung ab.
- Analoge Herleitung beim Anhängen von $0 \to$ Schlußvariable C

Übung 2.1

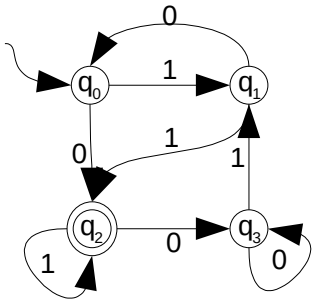
a) $\Sigma = \{0,1\}, \qquad 00, 100, 0101000, \ldots$



b) $\Sigma=\{0,1\}$, $000, 100010, \ldots$



Aufgabe 2.3

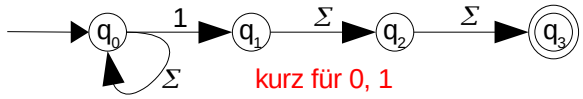


$V=\{q_0, q_1, q_2, q_3\}$
$q_0 \rightarrow 0\,q_2|0|1\,q_1|1$
$q_1 \rightarrow 0\,q_0|1\,q_2|1$
$q_2 \rightarrow 0\,q_3|1\,q_2|1$
$q_3 \rightarrow 0\,q_3|1\,q_1|1$

Aufgabe 2.4

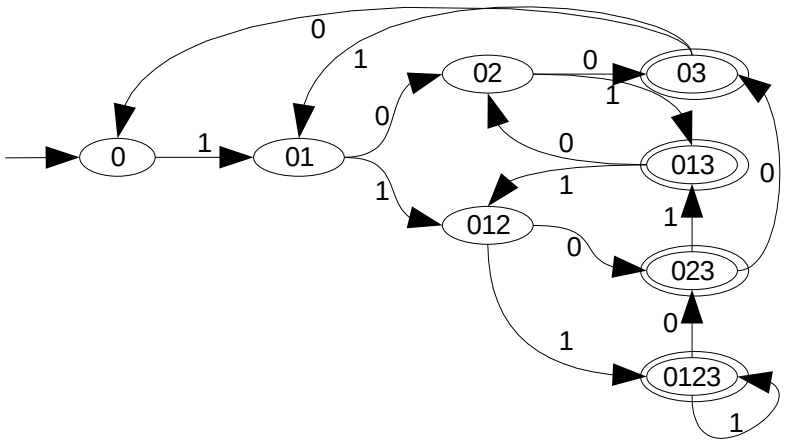
a) ...

b)

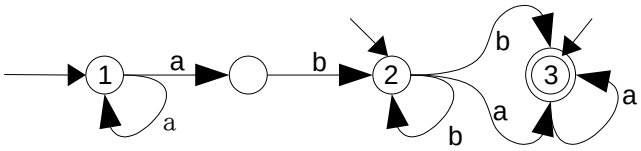


c)

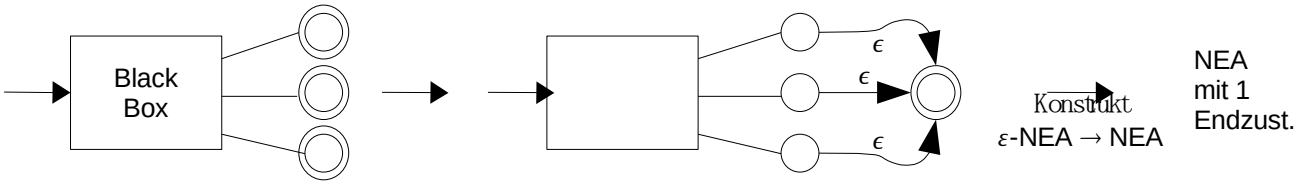
Q' / Σ	0	1
0	0	01
01	02	012
02	03	013 hallo :-)
012	023	0123
03	0	01
013	02	012
023	03	013
0123	023	0123



Aufgabe 2.6



Aufgabe 2.7



Aufgabe 2.8

a) $\Sigma^* 1 (0+1)(0+1)$
   $\Sigma^* 1 \Sigma \Sigma$

b) $0^* (0 0 0^* + 1)^* 0^*$

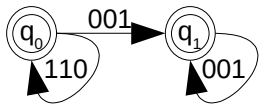
c) $(1 0 + 0)^* (1 0 + 1)^*$

Aufgabe 2.9

a) Binärwörter, bei denen im 1. Teil nur 1er-Gruppen gerader Länge, im 2. Teil nur 0er-Gruppen gerader Länge sind.

b) Binärwörter, bei denen maximal zwei 0en hintereinander stehen können.

Aufgabe 2.10



Aufgabe 2.14

	q_0	q_1	q_2	q_3
q_4	x	x	x	x
q_3	x		x	
q_2		x		
q_1	x			

Schritt 0

Schritt 1

$(0,3) \xrightarrow{0} (1,4)$ n.ä.

$(1,3) \xrightarrow{0} (4,4)$
$\xrightarrow{1} (2,0)$

$(2,3) \xrightarrow{0} (3,4)$ n.ä.

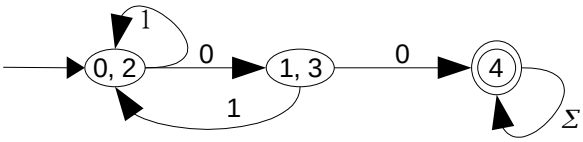
$(0,2) \xrightarrow{0} (1,3)$
$\xrightarrow{1} (2,2)$

$(1,2) \xrightarrow{0} (4,3)$ n.ä.

$(0,1) \xrightarrow{0} (1,4)$ n.ä.

Schritt 2: nochmal betrachten

keine Änderung



Aufgabe 2.16

n sei Konstante aus Pumping-Lemma.
Wir wählen $p \geq n$. $O^P = \alpha\beta\gamma$ sei die zugehörige Zerlegung.
Es sei $|\beta| := m > 1$.

$|\alpha\beta\gamma| = p$
$|\beta| = m$
$|\alpha\beta^i\gamma| = \ldots$ nicht prim
$= p + (i-1)m$
$\downarrow$
$= p + (p+1-1)m = p + pm = p(m+1)$ nicht prim!

Widerspruch! $\Rightarrow L$ nicht regulär!

Aufgabe 2.18

$L=\{0^l 1^m 0^{l+m} | l>0 \text{ und } m>0\}$

$|\omega| \geq n \rightarrow \omega = \alpha\beta\gamma$

1. $|\beta| \geq 1$
2. $|\alpha\beta| \leq n$
3. $\alpha\beta^i\gamma \in L \quad i=0,1,2,\ldots$

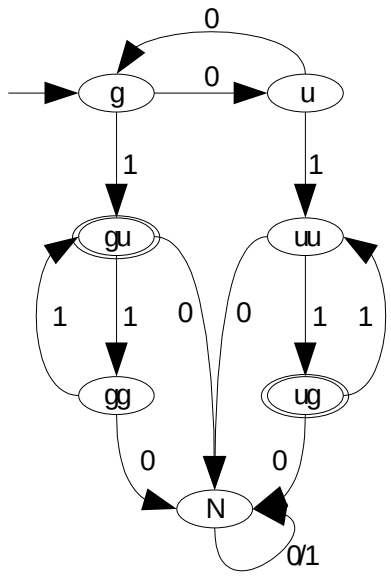
Wir wählen $l \geq n$ mit n der Konstante aus dem Pumping-Lemma. $\beta \neq \epsilon$ besteht dann nur aus 0en. Bei mehrfacher Wiederholung oder Weglassen von β ist die Anzahl der 0en hinter den 1en keinesfalls mehr gleich der Summe der vorderen 0en und 1en.

Aufgabe 2.25

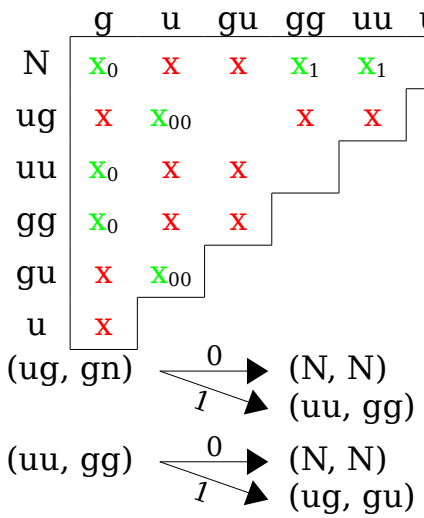
a) (00)* 1 (11)* + 0 (00)* (11)*

b) DEA dafür

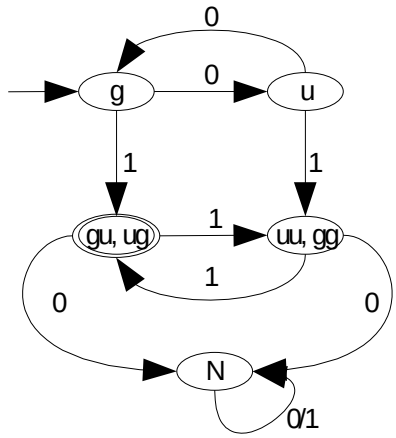
g (Start)	Gerade Anzahl 0en
u	Ungerade Anzahl 0en
gu	Gerade Anzahl 0en, ungerade Anzahl 1en
gg	Gerade Anzahl 0en, gerade Anzahl 1en
uu	Ungerade Anzahl 0en, ungerade Anzahl 1en
ug	Ungerade Anzahl 0en, gerade Anzahl 1en
N	Eingabe wird nicht akzeptiert



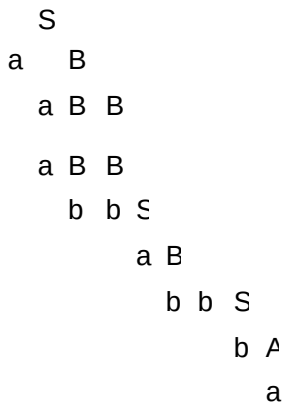
c)



$\Rightarrow ug \equiv gu\,,\quad uu \equiv gg$



Aufgabe 3.2



2 verschiedene Syntaxbäume, d.h. Grammatik ist mehrdeutig

a) $S \Rightarrow aB \Rightarrow aaBB \Rightarrow aaaBBB \Rightarrow aaabBB \Rightarrow aaabbSB \Rightarrow aaabbaBB \Rightarrow aaabbabbB \Rightarrow aaabbabbS \Rightarrow aaabbabbbA \Rightarrow aaabbabba$

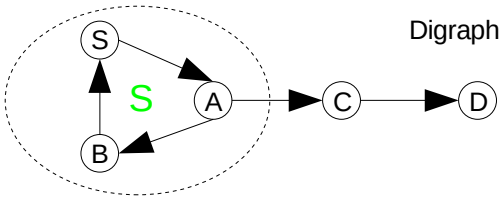
b) $S \Rightarrow aB \Rightarrow aaBB \Rightarrow aaBbS \Rightarrow aaBbbA \Rightarrow aaBbba \Rightarrow aaaBBbba \Rightarrow aaaBbSbba \Rightarrow aaaBbaBbba \Rightarrow aaaBbabbba \Rightarrow aaabbabbba$

Aufgabe 3.4

$S \to A|aB|aC$
$B \to S|Ba$
$D \to b|bDD$
$A \to B|C|cAb$
$C \to D|c$

Nutzlose Regeln: keine

Elimination Kettenregeln:



(a) Wir ersetzen B und A durch S (Zyklus!) und haben damit:

$S \to aS|aC|C|cSb|Sa$
$C \to D|c$
$D \to b|bDD$

(b) Von hinten nach vorne Kettenregeln eliminieren

Ersetze $C \to D$ durch $C \to b|bDD$ und
$S \to C$ durch $S \to b|bDD|c$

Regelmenge:

$S \to aS|aC|cSb|Sa|b|bDD|c$
$C \to c|b|bDD$
$D \to b|bDD$

Elimination gemischter rechter Seiten:

$S \to B_a S|B_a C|B_c SB_b|SB_a|b|B_b DD|c$
$C \to c|b|B_b DD$
$D \to b|B_b DD$
$B_a \to a$
$B_b \to b$
$B_c \to c$

Regeln mit mehr als zwei Variablen rechts:

$S \to B_a S|B_a C|B_c C_1|SB_a|b|B_b C_2|c$
$C_1 \to SB_b$
$C_2 \to DD$
$C \to c|b|B_b C_2$
$D \to b|B_b C_2$
$B_a \to a$
$B_b \to b$
$B_c \to c$

Aufgabe 3.6

Wahr.

„⇒“: Annahme: Sprache L erfüllt das Pumping-Lemma für REG.

Es existiert dann eine Zerlegung $\omega=\alpha\beta\gamma$ für $|\omega|\geq n$ mit

1. $|\beta|\geq 1$
2. $|\alpha\beta|\leq n$
3. $\alpha\beta^i\gamma\in L$

Wir setzen $\alpha':=\epsilon,\ \mu\beta'\nu=\alpha\beta$, mit $|\mu\nu|=|\beta|$ und $|\beta'|=|\alpha|$.

Behauptung: $\alpha'\mu\beta'\nu\gamma$ erfüllt Pumping-Lemma für CF.

1. $|\mu\nu|=|\beta|\geq 1$ ✓
2. $|\mu\beta'\nu|=|\alpha\beta|\leq n$ ✓
3. $\alpha'\mu^i\beta'\nu^i\gamma=\mu^i\beta'\nu^i\gamma\underset{*}{=}\beta'\mu^i\nu^i\gamma\underset{*}{=}\beta'(\mu\nu)^i\gamma=\alpha\beta^i\gamma$ * Alphabet ist einelementig

⇒ das Pumping-Lemma für CF ist erfüllt.

„⇐“: Annahme: L erfüllt Pumping-Lemma für CF.

Sei also $\omega=\alpha\mu\beta\nu\gamma$ eine entsprechende Zerlegung, die das Pumping-Lemma für CF erfüllt.

Behauptung: $\alpha'\beta'\gamma'$ mit $\alpha'=\beta,\ \beta'=\mu\nu,\ \gamma'=\alpha\gamma$ erfüllt Pumping-Lemma für REG.

1. $|\beta'|=|\mu\nu|\geq 1$ ✓
2. $|\alpha'\beta'|=|\beta\mu\nu|\underset{*}{=}|\mu\beta\nu|\leq n$ ✓
3. $\alpha'(\beta')^i\gamma'=\beta(\mu\nu)^i\alpha\gamma\underset{*}{=}\alpha\mu^i\beta\nu^i\gamma\in L$ ✓

⇒ das Pumping-Lemma für REG ist erfüllt.

Aus der Aussage folgt z.B., daß

$L=\{0^p|p\text{ist prim}\}$

nicht nur nicht-regulär ist (s.o.), sondern auch nicht kontextfrei.

Aufgabe 3.7

a) $\omega_1=aaaaa$

a	a	a	a	a
{A, C}	{A, C}	{A, C}	{A, C}	{A, C}
{B}	{B}	{B}	{B}	
{S, C, A}	{S, C, A}	{S, C, A}		
{B}	{B}			
{A, S, C}				

$\rightarrow\omega_1\in L(G)$

b) $\omega_2=aaaaaa$

a	a	a	a	a	a
{A, C}	{A, C}	{A, C}	{A, C}	{A, C}	{A, C}
{B}	{B}	{B}	{B}	{B}	
{S, C, A}	{S, C, A}	{S, C, A}	{S, C, A}		
{B}	{B}	{B}			
{A, S, C}	{A, S, C}				
{B}					

$\rightarrow \omega_2 \notin L(G)$

c) $\omega_3 = b\,a\,a\,b\,a$

b	a	a	b	a
{B}	{A, C}	{A, C}	{B}	{A, C}
{A, S}	{B}	{S, C}	{A, S}	
Ø	{B}	{B}		
Ø	{A, S, C}			
{S, C, A}				

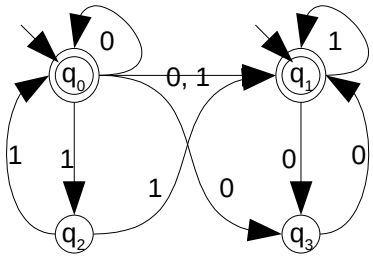
$\rightarrow \omega_3 \in L(G)$

Aufgabe zum CYK-Algorithmus

$S \rightarrow A\,B \mid B\,C \mid A\,C \mid D\,D \mid c$
$A \rightarrow A\,D \mid B\,A \mid a$
$B \rightarrow B\,A \mid C\,D \mid b$
$C \rightarrow A\,A \mid D\,B \mid c$
$D \rightarrow D\,B \mid A\,C \mid d$

c	d	b	a	a	c	d
{S, C}	{D}	{B}	{A}	{A}	{S, C}	{D}
{B}	{C, D}	{A, B}	{C}	{S, D}	{B}	
{B}	{C, D}	{C, A, B, S}	{A}	{S}		
{A, B}	{C, D}	{S, D, A, B}	{A}			
{C, A, B, S}	{B, S, C, D}	{S, A, B}				
{S, D, A, B}	{B, S, C, D}					
{S, ...}						

ε-NEA → NEA:

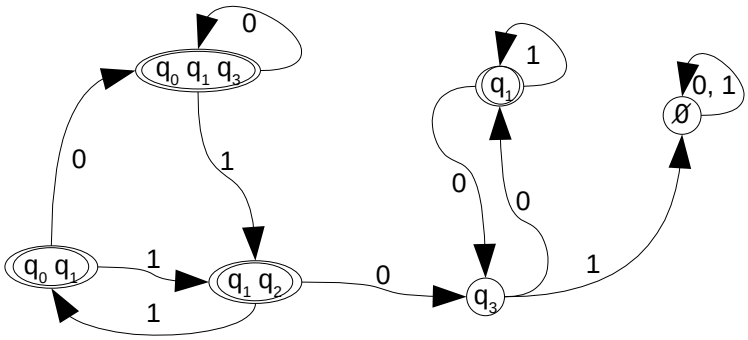


NEA → DEA:

δ':

	0	1
$q_0 q_1$	$q_0 q_1 q_3$	$q_1 q_2$
$q_0 q_1 q_2$	$q_0 q_1 q_3$	$q_1 q_2$
$q_1 q_2$	q_3	$q_0 q_1$
q_3	q_1	$\emptyset$
q_1	q_3	q_1
$\emptyset$	$\emptyset$	$\emptyset$

F'



Aufgabe 3.10

Idee: Wir speichern auf dem Keller, wie oft ein Zeichen mehr als das andere vorkam. Mit dem Zustand merken wir uns, welches Zeichen es ist:

q_0: überzählige 0en auf Keller
q_1: überzählige 1en auf Keller

$K=(\{q_0,q_1\},\{0,1\},\{0,\#\},\delta,q_0,\#)$

Wir akzeptieren in q_0:

$\delta(q_0,0,\#)=\{(q_0,a\#)\}$ $\delta(q_0,1,\#)=\{(q_1,a\#)\}$	Start
$\delta(q_0,0,a)=\{(q_0,aa)\}$	
$\delta(q_0,1,a)=\{(q_0,\epsilon)\}$	„normale“
$\delta(q_1,0,a)=\{(q_1,\epsilon)\}$	Verarbeitung
$\delta(q_1,1,a)=\{(q_1,aa)\}$	
$\delta(q_1,\epsilon,\#)=\{(q_0,\#)\}$	Bei (bis auf #) leeren Keller in q_0 wechseln (leerer Keller ≙ #0 = #1)

Akzeptierende Berechnung für ω_1=101001:

$(q_0,101001,\#)\vdash(q_1,01001,a\#)\vdash(q_1,1001,\#)\vdash(q_0,1001,\#)\vdash(q_1,001,a\#)$
$\vdash(q_1,10,\#)\vdash(q_0,01,\#)\vdash(q_0,1,a\#)\vdash(q_0,\epsilon,\#)$ ← akzeptierend

Aufgabe 4.4

DTM mit Startzustand q_0, Endzustand q_F:

1 0 1 0 1 0
0 1 0 0 1 1 0 0 0
0 1 0 0 0 1 0 0 nicht gültig

0er-Folgen werden immer länger

DTM mit Startzustand q_0, Endzustand q_F:

$\delta(q_0,\square)=(q_F,\square,\odot)$ leeres Eingabewort, Akzeptanz
$\delta(q_0,1)=(q_0,1,\rightarrow)$ erste 0 suchen
$\delta(q_0,0)=(q_1,X,\rightarrow)$ erste 0 löschen
$\delta(q_0,X)=(q_0,X,\rightarrow)$ erste 0 suchen (wenn bereits 0en gelöscht wurden)
$\delta(q_1,0)=(q_1,0,\rightarrow)$ 0en überspringen
$\delta(q_1,1)=(q_2,1,\rightarrow)$ Ende 0en Block
$\delta(q_1,\square)=(q_B,\square,\leftarrow)$ Iteration fertig
$\delta(q_2,1)=(q_2,1,\rightarrow)$ 1en überspringen
$\delta(q_2,0)=(q_1,X,\rightarrow)$ erste 0 Folgeblock löschen
$\delta(q_2,X)=(q_3,X,\rightarrow)$ auf den 1er-Block folgen Xe, es muß noch eine 0 kommen
$\delta(q_2,\square)=(q_B,\square,\leftarrow)$ Iteration fertig
$\delta(q_3,X)=(q_3,X,\rightarrow)$ suche nach 0
$\delta(q_3,0)=(q_1,X,\rightarrow)$ 0 löschen
$\delta(q_B,\begin{pmatrix}1\\0\\X\end{pmatrix})=(q_B,\begin{pmatrix}1\\0\\X\end{pmatrix},\leftarrow)$ löschen
$\delta(q_B,\square)=(q_1,\square,\rightarrow)$ neue Iteration starten

q_0: Start neue Iteration
q_1: Lösche erste 0 im nächsten 0en Block
q_2: 0en Block zu Ende, 1en überspringen
q_3: Es muß (nach Überspringen von Xen) eine 0 gelöscht werden
q_B: Rücklauf

Im Zustand q_3 auf $\square,1$ stoßen -> nicht akzeptieren

Aufgabe 6.2

Strategie: Zu jeder 0 vor der 1 auch eine 0 nach der 1 löschen

DTM:

$\delta(q_0,0)=(q_1,\square,\rightarrow)$ erste 0 links von 1 löschen
$\delta(q_1,0)=(q_1,0,\rightarrow)$ bis zur 1 laufen
$\delta(q_1,1)=(q_2,1,\rightarrow)$ bis zum Ende laufen
$\delta(q_2,0)=(q_2,0,\rightarrow)$ bis zum Ende laufen
$\delta(q_2,\square)=(q_3,\square,\leftarrow)$ zurück zur letzten 0
$\delta(q_3,0)=(q_4,\square,\rightarrow)$ letzte 0 löschen
$\delta(q_4,0)=(q_4,0,\leftarrow)$ zum Anfang laufen
$\delta(q_4,1)=(q_4,1,\leftarrow)$
$\delta(q_4,\square)=(q_0,\square,\rightarrow)$ von vorne beginnen
$\delta(q_3,1)=(q_5,0,\leftarrow)$ n0en gelöscht, aber vorne eine 0 zu wenig
$\delta(q_5,0)=(q_5,0,\leftarrow)$ SLK an Anfang positionieren
$\delta(q_5,\square)=(q_F,\square,\rightarrow)$ Fertig, q_F Endzustand
$\delta(q_0,1)=(q_6,\square,\rightarrow)$ $n\geq m,1$ löschen
$\delta(q_6,0)=(q_6,\square,\rightarrow)$ übrige 0en löschen
$\delta(q_6,\square)=(q_F,\square,\odot)$ Fertig

Aufgabe 6.4

```
          LOAD     #1   // z = 1
          STORE    3    // c(3) = 1
LOOP:     LOAD     2    // c(0) = m
          JZERO    END  // if (m == 0) ...  fertig
          SUB      #1   // m = m – 1
          STORE    2    // c(2) = m
          LOAD     3    // c(2) = z
          MULT     1    // z = z * n
          STORE    3    // c(3) = z
          GOTO     LOOP // nächste Iteration
END:      END
```