Theoretische Informatik II

Prof. Dr. Carsten Lutz
AG Theorie der künstlichen Intelligenz
MZH Raum 3090

Homepage der Vorlesung:
http://www.informatik.uni-bremen.de/tdki/lehre/ws09/theoinf

Organisatorisches

Vorlesung: Mo 10:00 – 12:00 MZH 1400

Hauptsächlich Folien,
ausgesuchte Beispiele + Beweise an der Tafel

Skript:

- Verfügbar auf Webseite
- Teile I + II: VL Theoretische Informatik I
- Teile III + IV: VL Theoretische Informatik II
- Zusätzliches Material in der VL wird angekündigt

  Mitschreiben!

Literatur

- Skript zur Vorlesung (Webseite)
- Dexter Kozen, Automata and Computability, Springer Verlag 2007
- John Hopcroft, Rajeev Motwani, Jeff Ullmann, Introduction to Automata Theory, Languages, and Computation (3rd edition), Addison Wesley, 2006
- Uwe Schöning, Theoretische Informatik-kurzgefasst, Spektrum Akademischer Verlag, 2001
- Ingo Wegener, Theoretische Informatik-Eine algorithmenorientierte Einführung, Teubner, 1999.

Übungsgruppen

- 6 Gruppen zu unterschiedlichen Terminen, Eintragen ab 15:00 über Stud.IP (nur eine Gruppe!)
- Beginn kommende Woche
- Jede Woche ein Aufgabenblatt auf VL-Homepage, das in der Übungsgruppe gemeinsam gelöst wird
- Jede zweite Woche werden die Aufgaben abgegeben und korrigiert (bis Montags 24:00, Postfach Tutor)
- Die Bearbeitung der Aufgaben erfolgt in Gruppen von 2-3 Personen
- In der kommenden Woche muss nichts abgegeben werden.

Scheine / Prüfungen

Prüfungsmodalitäten

- Auf den ersten 3 und den letzten 3 Übungsblättern müssen 50% der Punkte erreicht werden

  Beispiel 1: 30, 50, 70 + 0, 100, 90 = ok

  Beispiel 2: 80, 90, 70 + 0, 100, 40 = nicht ok

- Note wird über alle Blätter gemittelt
- Zusätzlich Fachgespräch am Ende des Semesters (Prüfungsleistung, Änderung der Note möglich)

Einführung

Alphabet: endliche Menge von Symbolen, z.B. $\Sigma = \{a, b, c\}$

Wort über Alphabet Σ: endliche Folge von Symbolen aus Σ, z.B.:

$$w_1 = aabcaabbcbaa \qquad w_2 = abba \qquad w_3 = \varepsilon$$

Formale Sprache über Alphabet Σ: Menge von Wörtern über Σ, z.B.:

$$L_1 = \{a^n b^n \mid n \geq 0\} = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$$

$$L_2 = \{a^n \mid n \text{ prim } \} = \{aa, aaa, aaaaa, aaaaaaa, \ldots\}$$

$$L_3 = \emptyset$$

$$L_4 = \{\varepsilon\}$$

Interessante Sprachen sind i.d.R. unendlich.

Grammatiken sind allgemeines Werkzeug zur Definition von Sprachen

Zum Beispiel folgende Grammatik G:

$$\begin{aligned} S &\longrightarrow aSb \\ S &\longrightarrow \varepsilon \end{aligned}$$

Ableitbare Wörter: ε, ab, $aabb$, $aaabbb$, $\ldots$

Erzeugte Sprache also $L(G) = \{a^n b^n \mid n \geq 0\}$

Verschiedene Typen von Grammatiken charakterisiert durch die Form von Regeln

Komplexere Typen von Grammatiken können komplexere Sprachen erzeugen.

Definition (Chomsky-Hierarchie, Typen von Grammatiken)

Eine Grammatik $G = (N, \Sigma, P, S)$ ist vom...

Typ 0 Jede Grammatik

Typ 1 Kontextsensitive Grammatik:

Regeln $w \longrightarrow u$ mit $|u| \geq |w|$ (nicht verkürzend)

Ausnahme: Regel $S \longrightarrow \varepsilon$ erlaubt

wenn S nicht auf der rechten Seite einer Produktion vorkommt.

Typ 2 Kontextfreie Grammtik:

Regeln $A \longrightarrow w$ mit A Nichtterminal, w beliebig.

Typ 3 Rechtslineare Grammatik:

Regeln $A \longrightarrow uB$ oder $A \longrightarrow u$ mit A Nichtterminal, u Terminalwort

Beispiel 6.3 (Fortsetzung)

$G = (N, \Sigma, P, S)$ mit $N = \{S, B\}$, $\Sigma = \{a, b, c\}$ und

$$P = \{S \longrightarrow aSBc,\ S \longrightarrow abc,\ cB \longrightarrow Bc,\ bB \longrightarrow bb\}$$

$$S \vdash_G abc$$

$$S \vdash_G aSBc \vdash_G aaSBcBc \vdash_G aaabcBcBc \vdash_G aaabBccBc$$

$$\vdash_G^2 aaabBBccc \vdash_G^2 aaabbbccc$$

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

Diese Sprache ist unsere ‘prototypische’ kontextsensitive Sprache.

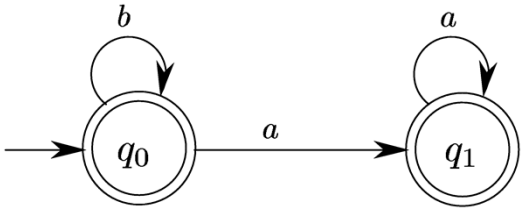
Wir hatten bereits gezeigt:

sie ist nicht kontextfrei! (Pumping Lemma für kontextfreie Sprachen)

Automaten sind weiteres wichtiges Werkzeug zur Definition von Sprachen

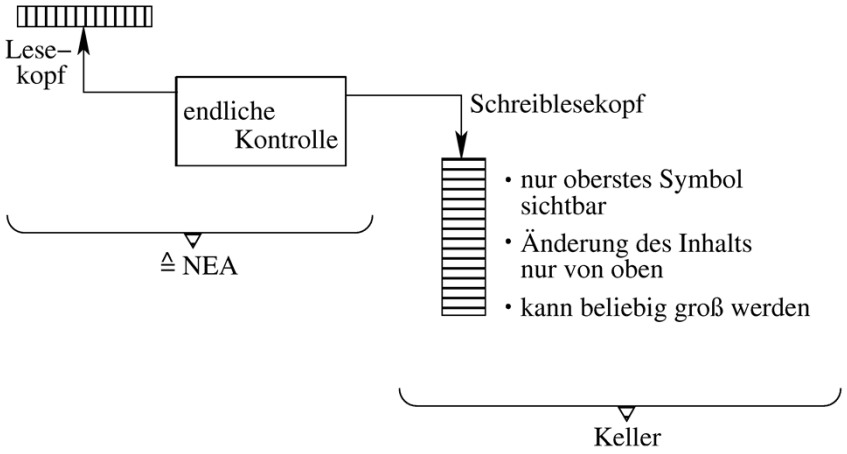
Endliche Automaten erkennen genau die regulären Sprachen (Typ 3)

z.B.

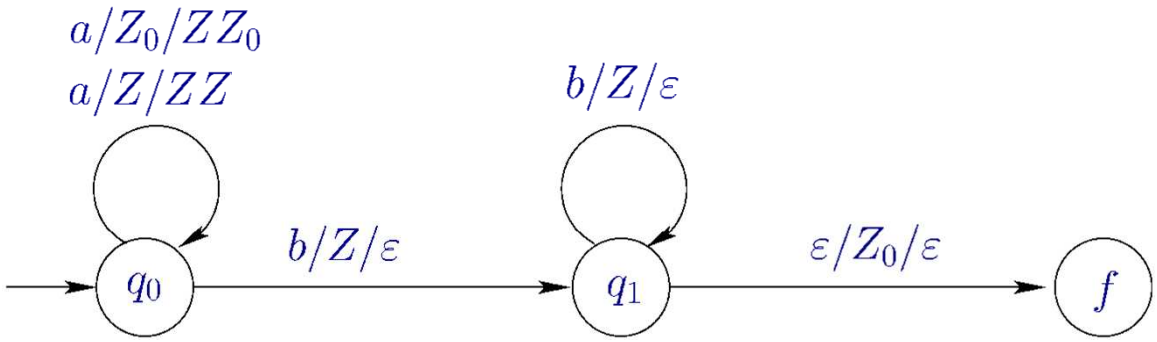


$L(A) = b^*a^*$

Kellerautomaten erkennen genau die kontextfreien Sprachen (Typ 2)



Konkreter Kellerautomat A:



Es gilt $L(A) = \{a^n b^n \mid n \geq 1\}$ (unsere 'prototypische' kontextfreie Sprache):

- für jedes gelesene a wird ein Z auf den Keller geschoben,
  für jedes gelesene b ein Z vom Keller genommen
- ist die Eingabe nicht aus a^*b^*, so blockiert der Automat
- wenn nach Lesen der Eingabe das Kellerstartsymbol oben auf Keller liegt,
  akzeptiert A (per leerem Keller)

Weitere wichtige Themen: Abschlusseigenschaften

	$L_1 \cap L_2$	$L_1 \cup L_2$	$\overline{L}$	$L_1 \cdot L_2$	L^*
kontextfrei	✗	✓	✗	✓	✓
regulär	✓	✓	✓	✓	✓

...und die algorithmische Lösbarkeit und Komplexität wichtiger Probleme:

	Wortproblem	Leerheitsprob..	Äquivalenzprob.
Typ 2 Grammatik / PDA	polyzeit	polyzeit	???
NEA / reg. Ausdruck / Typ 3 Grammatik	linearzeit	linearzeit	wohl nicht polyzeit
DEA	linearzeit	linearzeit	polyzeit

Theoretische Informatik I

Zwei Sichten auf Theoretische Informatik II:

Sicht 1: Studium der Sprachen vom Typ 0 und vom Typ 1

- Wie sehen geeignete Automatenmodelle aus?
- Welche Abschlusseigenschaften werden erfüllt?
- Wie schwierig sind das Wort-/Leerheits-/Äquivalenzproblem?

Sicht 2: Wir studieren fundamentale Fragen der Berechenbarkeit und Komplexität

- Gibt es Probleme, die prinzipiell nicht berechenbar sind?
- Sind alle Berechnungsmodelle gleich mächtig?
  (verschiedene Rechnerarchitekturen, Programmiersprachen, abstrakte mathematische Modelle)
- Wenn ein Problem berechenbar ist, welche Ressourcen (Zeit und Speicher) benötigt man mindestens?

Berechenbarkeit

Berechenbarkeit

Stelligkeit

Wir interessieren uns für Funktionen $f : \mathbb{N}^k \to \mathbb{N}$ bzw. $f : (\Sigma^*)^k \to \Sigma^*$

(wir werden sehen: das ist quasi dasselbe / wechselseitig "übersetzbar")

Zum Beispiel:

- Konstante Nullfunktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(x) = 0$
- Binäre Additionsfunktion $f : \mathbb{N}^2 \to \mathbb{N}$ mit $f(x, y) = x + y$
- Binäre Konkatenation von Wörtern $f : (\Sigma^*)^2 \to \Sigma^*$ mit $f(v, w) = vw$

Intuitiv nennen wir f berechenbar wenn es einen Algorithmus gibt,
der f berechnet, also:

bei Eingabe $(x_1, \ldots, x_n)$ Ausgabe $f(x_1, \ldots, x_n)$ (nach endlicher Zeit!)

Berechenbarkeit

Fundamentale Beobachtung:
Es gibt Funktionen, die wohldefiniert und vollständig beschrieben sind, aber trotzdem nicht berechenbar.

Nicht klar ist z.B. die Berechenbarkeit der Funktion $f : \mathbb{N} \to \mathbb{N}$ mit

$$f(n) = \begin{cases} 1 & \text{wenn } n \text{ als “Teilwort” in der Dezimaldarstellung der Zahl } \pi \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$

Zum Beispiel $f(1415) = 1$, da $\pi = 3,1415...$

Dieses f ist wohldefiniert und vollständig beschrieben, aber es ist nicht direkt klar, wie man einen f berechnenden Algorithmus findet

(es ist nicht ausreichend, nach und nach alle Dezimalstellen von π zu erzeugen und einfach auf das Teilwort n “zu warten”: Endlosschleife wenn $f(n) = 0$!)

Berechenbarkeit

Nachweis der Berechenbarkeit:

Um für eine Funktion f intuitiv klarzumachen, dass sie berechenbar ist, genügt es, einen Algorithmus anzugeben, der f berechnet.

Dies kann z.B. in Form eines Pascal-, Java- oder C-Programms geschehen oder als abstrakte Beschreibung der Vorgehensweise.

Wir haben in Theorie I einige Berechenbarkeitsbeweise geführt, z.B. das Wortproblem für kontextfreie Sprache L:

- abstrakte Beschreibung: Wandel einer Typ 2 Grammatik in Chomsky-NF
- CYK-Algorithmus in Pseudocode

Genug Information für Implementierung in konkreter Programmiersprache!

Wir haben also eher eine intuitiven Begriff von Berechenbarkeit verwendet!

Aber wie beweist man, dass für ein Problem kein Algorithmus existiert?

Dazu muss man zunächst formal (statt intuitiv!) die Frage beantworten:

Was ist ein Algorithmus?

Mögliche Antworten:

- Programmiersprachen: C, Pascal, Java, Lisp, Prolog, Assembler, etc.
- Mathematische Formalismen: Turingmaschine, Registermaschine (RAM), WHILE Programme, μ-berechenbare Funktionen, λ-Kalkül, Abstract State Machines (ASMs), etc.

Fundamental: alle diese Modelle sind gleich mächtig, d.h.: jede Funktion ist entweder in allen diesen Modellen berechenbar oder in keinem davon

Wir werden in dieser Vorlesung drei Berechnungsmodelle betrachten:

- Turingmaschinen
- WHILE-Programme
- μ-rekursive Funktionen

} gleich mächtig

Church-Turing These:

> Die (intuitiv) berechenbaren Funktionen sind genau die mit Turingmaschinen (und damit auch mit μ-rekursiven Funktionen, WHILE-Programmen, Java-Programmen etc.) berechenbaren Funktionen.

Man spricht hier von einer These und nicht von einem Satz, da es nicht möglich ist, diese Aussage formal zu beweisen:

der intuitive Berechenbarkeitsbegriff ist ja nicht formal definierbar,
man kann darüber also keine formalen Beweise führen.

Es gibt aber sehr gute Indizien für die Richtigkeit der These:

Vielzahl äquivalenter Berechnungsmodelle.

Teil III: Berechenbarkeit

§11. Turingmaschinen

§12. Zusammenhang zwischen Turingmaschinen und Grammatiken

§13. Berechenbarkeit, Entscheidbarkeit, Aufzählbarkeit und Zusammenhänge

§14. Primitiv rekursive Funktionen und LOOP-Programme

§15. μ-rekursive Funktionen und WHILE-Programme

§16. Universelle Turingmaschinen und unentscheidbare Probleme

§17. Weitere unentscheidbare Probleme

Teil IV: Komplexität

§18. Komplexitätsklassen

§19. NP-vollständige Probleme

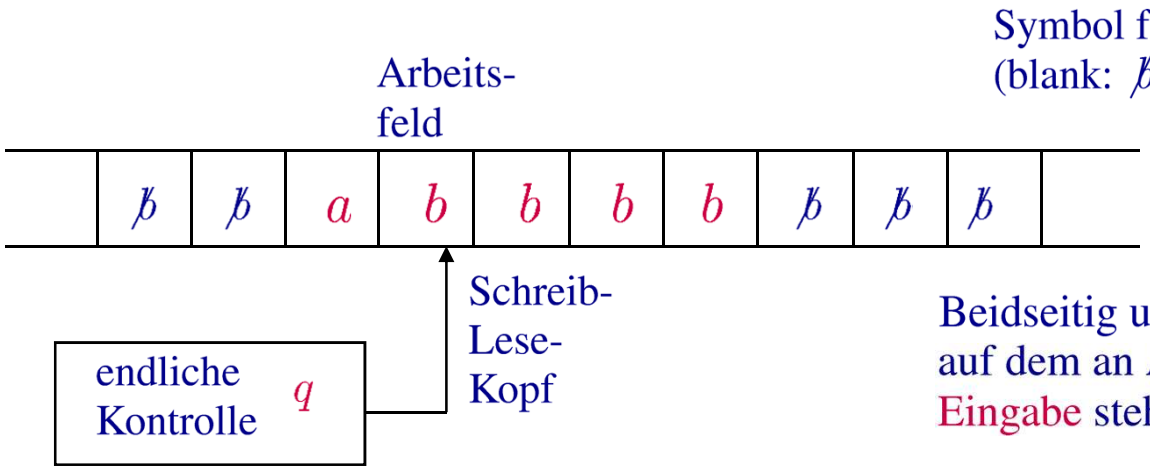
Turingmaschine:

- entwickelt 1936 von Alan Turing, um Berechenbarkeit zu studieren
- Sehr einfaches Berechnungsmodell, gut handhabbar in Beweisen
  ("It is amazing how little you need to have everything")
- Ähnlichkeiten zu endlichen Automaten und Kellerautomaten

Zwei Anwendungen:

- Formalisierung von "Algorithmus'' bei Nachweis der Nicht-Berechenbarkeit von Funktionen
- liefert Automatenmodell für Sprachen vom Typ 0 und, in eingeschränkter Form, auch für Sprachen vom Typ 1

§ 11. Turing-Maschinen



Zu jedem Zeitpunkt sind nur endlich viele Symbole auf dem Band verschieden von ß.

Definition 11.1 (Turingmaschine)

Eine Turingmaschine über dem Eingabealphabet Σ hat die Form $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$, wobei

- Q endliche Zustandsmenge ist,
- Σ das Eingabealphabet ist,
- Γ das Arbeitsalphabet ist mit $\Sigma \subseteq \Gamma$, $\not b \in \Gamma \setminus \Sigma$,
- $q_0 \in Q$ der Anfangszustand ist,
- $F \subseteq Q$ die Endzustandsmenge ist und
- $\Delta \subseteq Q \times \Gamma \times \Gamma \times \{r, l, n\} \times Q$ die Übergangsrelation ist.

Bedeutung der Übergangsrelation $\Delta \subseteq Q \times \Gamma \times \Gamma \times \{r, l, n\} \times Q$

$(q, a, a', \begin{matrix} r \\ l \\ n \end{matrix}, q') \in \Delta$ sagt:

- Im Zustand q
- mit a auf dem gerade gelesenen Feld (Arbeitsfeld)

kann die Turing-Maschine $\mathcal{A}$ folgende Operation ausführen:

- Symbol a durch a' ersetzen,
- in den Zustand q' wechseln
- den Schreib-Lesekopf um ein Feld nach rechts (r), links (l) oder nicht (n) bewegen.

Deterministische und nicht-deterministische Turing-Maschinen:

Die Maschine $\mathcal{A}$ heißt deterministisch, falls es für jedes Tupel $(q, a) \in Q \times \Gamma$ höchstens ein Tupel der Form $(q, a, \ldots, \ldots, \ldots) \in \Delta$ gibt.

NTM steht im folgenden für (möglicherweise) nicht-deterministische Turing-maschinen und DTM für deterministische.

Bei einer DTM gibt es zu jedem Berechnungszustand höchstens einen Folge-zustand, während es bei einer NTM mehrere geben kann.

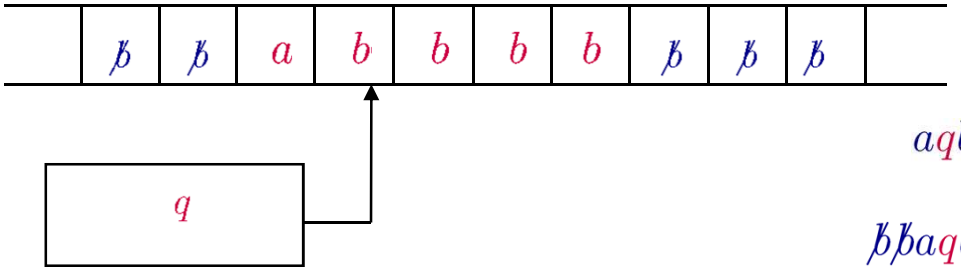
Berechnungszustand (Konfiguration) einer Turingmaschine:

kann man beschreiben durch ein Wort $\alpha q \beta$ mit $\alpha, \beta \in \Gamma^+$, $q \in Q$:

- q ist der momentane Zustand
- α ist die Beschriftung des Bandes links vom Arbeitsfeld
- β ist die Beschriftung des Bandes beginnend beim Arbeitsfeld nach rechts

Dabei werden (um endliche Wörter α, β zu erhalten) unendlich viele Blanks weggelassen, d.h. α, β umfassen mindestens den Bandabschnitt, auf dem Symbole $\neq \not b$ stehen.

Beispiel:



Die Übergangsrelation Δ ermöglicht folgende Konfigurationsübergänge:

Relation $\vdash_{\mathcal{A}}$ auf der Menge aller Konfigurationen:

es seien $\alpha, \beta \in \Gamma^+,\ \beta' \in \Gamma^*,\ a, b, a' \in \Gamma,\ q, q' \in Q$

- $\left.\begin{array}{rcl} \alpha q a \beta & \vdash_{\mathcal{A}} & \alpha a' q' \beta \\ \alpha q a & \vdash_{\mathcal{A}} & \alpha a' q' \not{b} \end{array}\right\}$ falls $(q, a, a', r, q') \in \Delta$

- $\left.\begin{array}{rcl} \alpha b q a \beta' & \vdash_{\mathcal{A}} & \alpha q' b a' \beta' \\ b q a \beta' & \vdash_{\mathcal{A}} & \not{b}\, q' b a' \beta' \end{array}\right\}$ falls $(q, a, a', l, q') \in \Delta$

- $\alpha q a \beta' \vdash_{\mathcal{A}} \alpha q' a' \beta'$ falls $(q, a, a', n, q') \in \Delta$

Folgekonfiguration:

Gilt $k \vdash_{\mathcal{A}} k'$, so heißt k' Folgekonfiguration von k.

Akzeptierende Konfiguration:

Die Konfiguration $\alpha q \beta$ heißt akzeptierend, falls $q \in F$.

Stoppkonfiguration:

Die Konfiguration $\alpha q \beta$ heißt Stoppkonfiguration, falls sie keine Folgekonfiguration hat.

Berechnung von $\mathcal{A}$:

Endliche oder unendliche Konfigurationsfolge $k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} k_2 \vdash_{\mathcal{A}} \cdots$

Für DTMs gibt es nur eine maximale Berechnung ab Konfiguration k_0, für NTMs kann es mehrere geben.

Endlicher Automat macht exakt so viele Schritte wie die Eingabe lang ist.

Eine wesentliche Eigenschaft von Turingmaschinen ist, dass sie unendlich lange weiterlaufen können, also nicht terminieren.

Triviales Beispiel: Turingmaschine mit

- Eingabealphabet $\Sigma = \{a\}$ $(= \Gamma)$
- Zustandsmenge $Q = \{q_0\}$ und Endzuständen $F = \emptyset$
- einzigem Übergang (q_0, a, a, n, q_0)

Unendliche Berechnung z.B.:

$$\not{b}q_0a \vdash_{\mathcal{A}} \not{b}q_0a \vdash_{\mathcal{A}} \not{b}q_0a \vdash_{\mathcal{A}} \cdots$$

Zwei Anwendungen:

- Berechnen von partieller Funktion $f : (\Sigma^*)^n \to \Sigma^*$

  Verwendung von DTM

  Eingabe $(w_1, \ldots, w_n)$ repräsentiert als $w_1 \not b w_2 \not b \cdots \not b w_n$

  Ende der Berechnung in Stoppkonfiguration

  Ausgabe ab Kopfposition bis zu erstem nicht-Σ-Symbol

- Erkennen von Sprache L

  Verwendung von DTM oder NTM

  Eingabe ab Kopfposition bis zu erstem $\not b$

  Akzeptanz wenn es Lauf gibt, der in akzeptierender Stoppkonfiguration endet

  Sonst wird Eingabe verworfen

Definition 11.2a (Turing-berechenbar)

Die partielle Funktion

$$f : (\Sigma^*)^n \to \Sigma^*$$

heißt Turing-berechenbar, falls es eine DTM $\mathcal{A}$ gibt, für die gilt:

- der Definitionsbereich $\mathrm{dom}(f)$ von f besteht aus genau den Tupeln $(w_1, \ldots, w_n) \in (\Sigma^*)^n$ so dass $\mathcal{A}$ ab Konfiguration

  $$k_0 = \not{b}\, q_0 w_1 \not{b} \ldots \not{b}\, w_n \not{b}$$

  eine Stoppkonfiguration erreicht;

- wenn $(w_1, \ldots, w_n) \in \mathrm{dom}(f)$, dann hat die von k_0 aus erreichte Stopp- konfiguration k die Form $k = uqxv$ mit
  - $x = f(w_1, \ldots, w_n)$ und
  - $v \in (\Gamma \setminus \Sigma) \cdot \Gamma^* \cup \{\varepsilon\}$.

Beachte:

1. Wir verwenden partielle Funktionen, da Turingmaschinen nicht anhalten müssen; für manche Eingaben ist der berechnete Funktionswert also nicht definiert

2. Wir erlauben hier nur deterministische Maschinen, da sonst der Funktionswert nicht eindeutig sein müßte.

3. Funktionen $(\Sigma^*)^n \to \Sigma^*$ mit $|\Sigma| = 1$ kann man auch als Funktionen $\mathbb{N}^n \to \mathbb{N}$ auffassen:

   Ein-/Ausgabe a^k repräsentiert Zahl k, unär kodiert

   Wir unterscheiden im folgenden nicht immer explizit zwischen beiden Arten von Funktionen

Beispiel 11.3

Die Funktion

$$f : \mathbb{N} \to \mathbb{N} : \quad n \mapsto 2n$$

ist Turing-berechenbar.

Wie kann eine Turingmaschine die Anzahl der as auf dem Band verdoppeln?

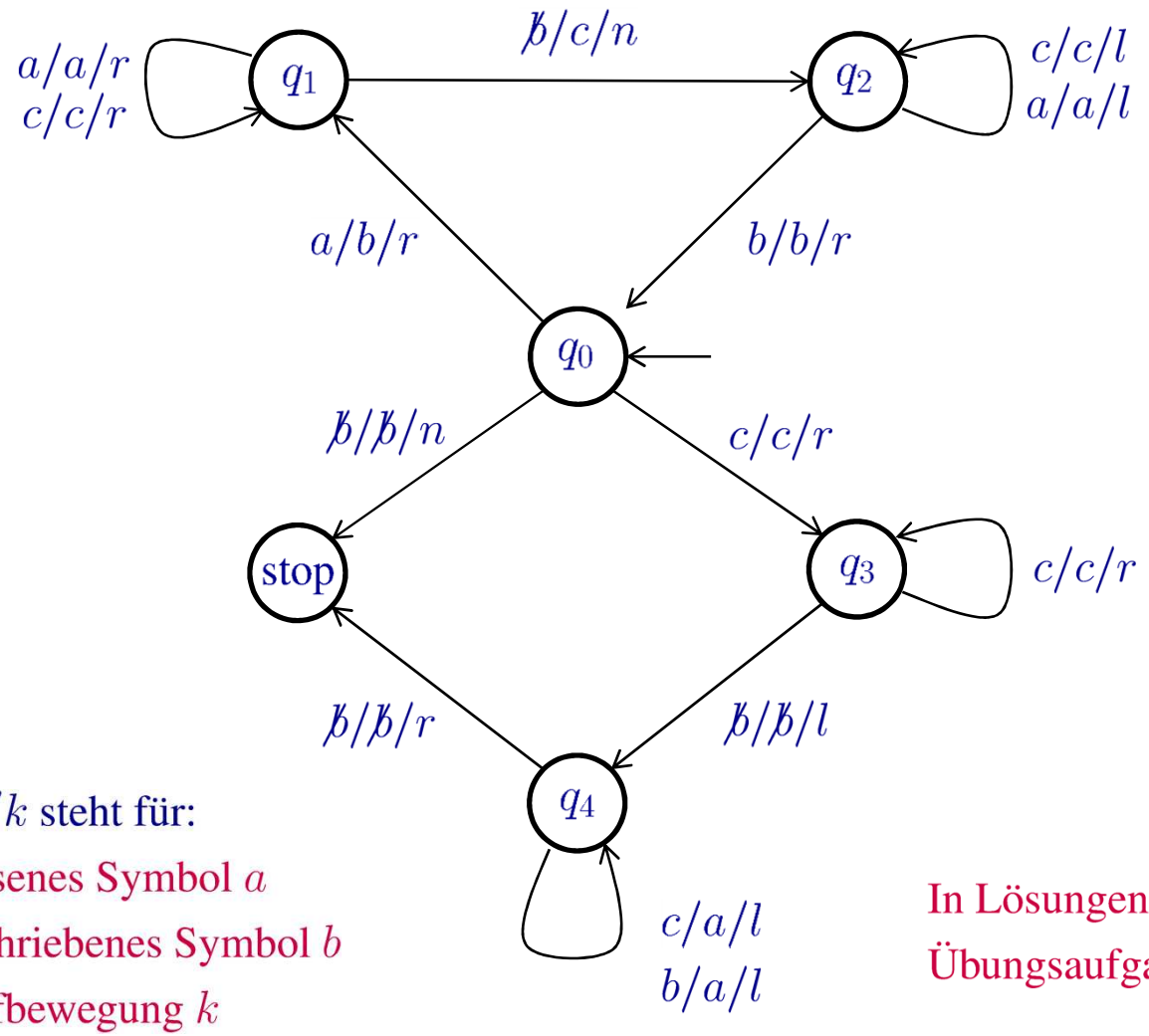
Idee:

- Ersetze das erste a durch b,
- laufe nach rechts bis zum ersten Blank und ersetze dieses durch c
- laufe zurück bis zum zweiten a (unmittelbar rechts vom b) und ersetze dieses durch b,
- laufe nach rechts bis zum ersten Blank etc.
- Sind alle as aufgebraucht, so ersetze noch die bs und cs wieder durch as.

Dies wird durch die folgende Übergangstafel realisiert:

q_0	$\not b$	$\not b$	n	stop	$2 \cdot 0 = 0$
q_0	a	b	r	q_1	ersetze a durch b $\quad (\star)$
q_1	a	a	r	q_1	laufe nach rechts über as
q_1	c	c	r	q_1	und bereits geschriebene cs
q_1	$\not b$	c	n	q_2	schreibe weiteres c
q_2	c	c	l	q_2	laufe zurück über cs und
q_2	a	a	l	q_2	as
q_2	b	b	r	q_0	bei erstem b eins nach rechts und weiter wie $(\star)$ oder
q_0	c	c	r	q_3	alle as bereits ersetzt
q_3	c	c	r	q_3	laufe nach rechts bis Ende der cs
q_3	$\not b$	$\not b$	l	q_4	letztes c erreicht
q_4	c	a	l	q_4	ersetze cs und bs
q_4	b	a	l	q_4	durch as
q_4	$\not b$	$\not b$	r	stop	bleibe am Anfang der erzeugten $2n$ as stehen

Grafische Darstellung der eben entworfenen Turingmaschine:



$a/b/k$ steht für:

gelesenes Symbol a

geschriebenes Symbol b

Kopfbewegung k

In Lösungen für Übungsaufgaben verwenden!

Definition 11.2b (Akzeptierte Sprache, Turing-akzeptierbar)

Die von einer NTM $\mathcal{A}$ akzeptierte Sprache ist

$$L(\mathcal{A}) := \{w \in \Sigma^* \mid \not{b}\, q_0 w \not{b} \vdash_{\mathcal{A}}^* k, \text{ wobei } k \text{ akzeptierende Stoppkonfiguration ist}\}$$

Die Sprache $L \subseteq \Sigma^*$ heißt Turing-akzeptierbar, falls es eine NTM $\mathcal{A}$ gibt mit $L = L(\mathcal{A})$.

Beachte:

$= \text{beginnend in } \not{b}q_0w$

Wenn $w \notin L(\mathcal{A})$, dann gilt für jede Berechnung von $\mathcal{A}$ auf w:

- die Berechnung endet in nicht-akzeptierender Stoppkonfiguration oder
- die Berechnung ist unendlich.

Beispiel 11.4

Die Sprache

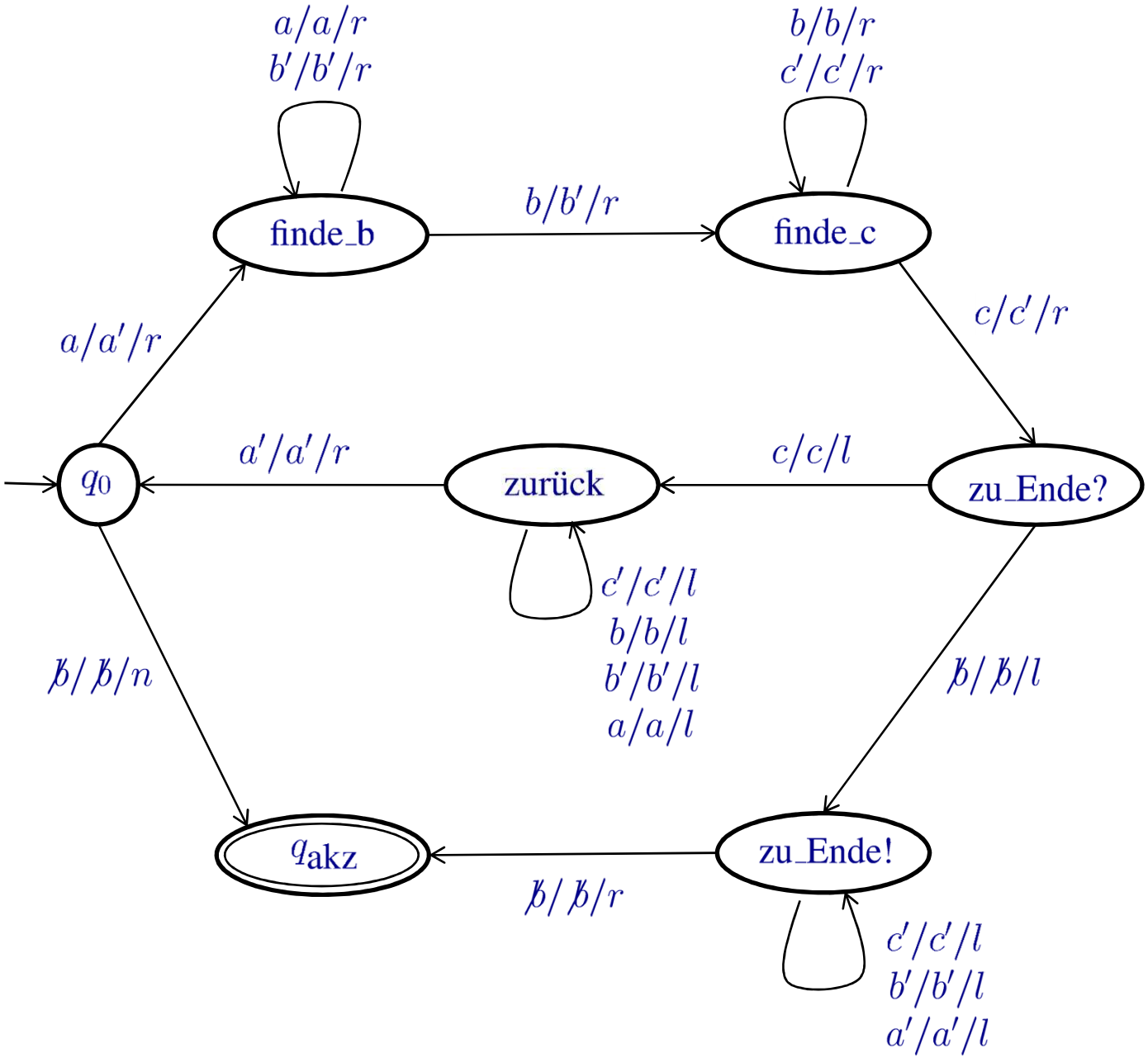
$$L = \{a^n b^n c^n \mid n \geq 0\}$$

ist Turing-akzeptierbar.

Die Turingmaschine, welche L akzeptiert, geht wie folgt vor:

- Sie ersetzt erstes a durch a', erstes b durch b' und erstes c durch c';
- läuft zurück und wiederholt diesen Vorgang
- Falls ein a rechts von einem b oder c steht, verwirft die TM
  Ebenso, wenn ein b rechts von einem c steht.

  Stopp in Nicht-Endzustand
- Dies wird solange gemacht, bis nach erzeugtem c' ein $\not b$ steht.
- Zum Schluß wird zurückgelaufen und überprüft, dass keine unersetzten a oder b übrig geblieben sind.


$a/a/r$
$b'/b'/r$
$b/b/r$
$c'/c'/r$
finde_b
$b/b'/r$
finde_c
$a/a'/r$
$c/c'/r$
q_0
$a'/a'/r$
zurück
$c/c/l$
zu_Ende?
$c'/c'/l$
$b/b/l$
$b'/b'/l$
$a/a/l$
$\not b/\not b/n$
$\not b/\not b/l$
q_{akz}
zu_Ende!
$\not b/\not b/r$
$c'/c'/l$
$b'/b'/l$
$a'/a'/l$