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/ss13/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

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

- 7 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 12:00, Postfach Tutor)
- Die Bearbeitung der Aufgaben erfolgt in Gruppen von 2-3 Personen
- In der kommenden Woche muss nichts abgegeben werden.

Scheine / Prüfungen

Prüfungsmodalitäten

- Insgesamt müssen 50% der Punkte erreicht werden
- Note wird über alle Blätter gemittelt
- Zusätzlich Fachgespräch am Ende des Semesters (Prüfungsleistung, Änderung der Note möglich)
- Übungen gehören zum Vorlesungsstoff!

Theoretische Informatik II

Behandelte Gebiete: Berechenbarkeit und Komplexität

Kann jede Programmiersprache dasselbe berechnen?

Gibt es Probleme, die gar nicht berechenbar sind /
was sind die Grenzen der Berechenbarkeit?

Sind alle Berechungsprobleme gleich schwer?

Kann man effizient zwei Zahlen addieren?

Kann man effizient Sudokus lösen?

Was heißt “effizient”?

Zusammenhang zu Theoretische Informatik 1

Unser zentrales Berechnungsmodell werden Turingmaschinen sein

Dadurch ergibt sich ein Zusammenhang zu den formalen Sprachen:

- Turingmaschinen stellen sich als “richtiges” Automatenmodell für Typ 0-Sprachen heraus
- in modifizierter Form liefern sie Automatenmodell für Typ 1-Sprachen

Sprachen vom Typ 0 gehören also ins Gebiet der Berechenbarkeit

Die in TheoInf I gestellten Fragen (Abschlusseigenschaften, Wortproblem, etc) haben in der Berechenbarkeit oft eine natürliche Bedeutung.

Theoretische Informatik 1 – Kurze Wiederholung

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

Formal ist Grammatik ein 4-Tupel $G = (N, \Sigma, P, S)$

Zum Beispiel:

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

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

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

$$S \vdash_G abc$$

$$\underline{S} \vdash_G \underline{a\overline{S}Bc} \vdash_G \underline{a\overline{aS}BcBc} \vdash_G aa\underline{abc}\overline{B}cBc \vdash_G aaab\underline{\overline{B}cc}Bc$$

$$\vdash_G^2 aaa\overline{bB}\underline{Bccc} \vdash_G^2 aaa\overline{bbb}ccc$$

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

Diese Sprache ist unsere ‘prototypische’ kontextsensitive Sprache.

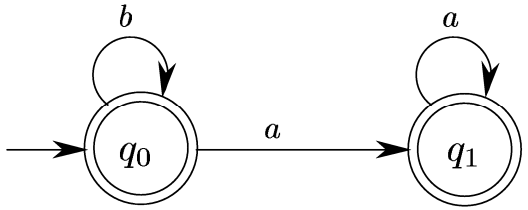
Wir hatten bereits gezeigt:

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

Automaten sind weiteres wichtiges Werkzeug zur Definition von Sprachen

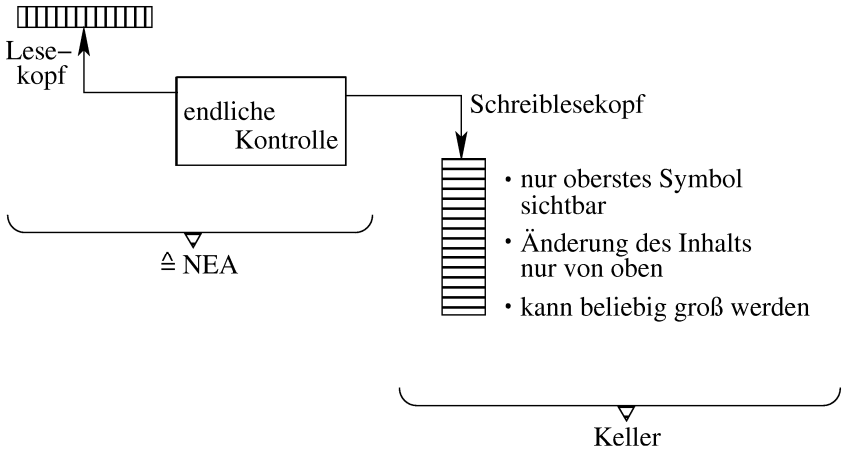
Endliche Automaten erkennen genau die regulären Sprachen (Typ 3)

z.B.

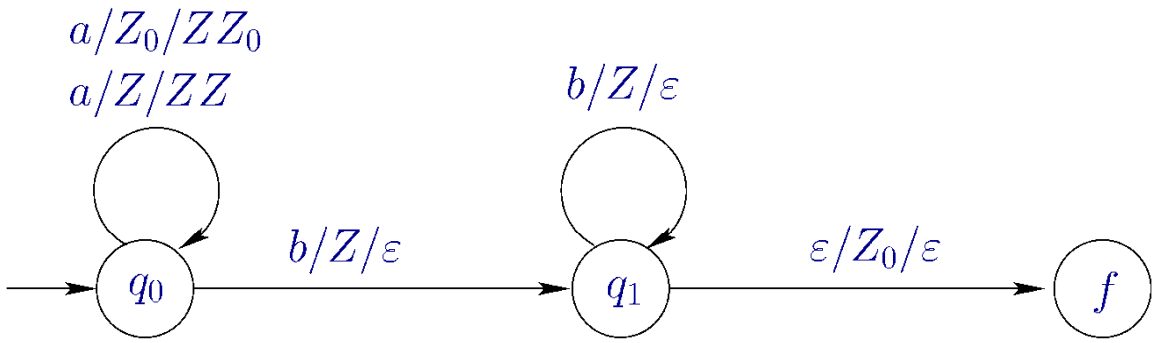


$L(A) = b^*a^*$

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

Einführung Berechenbarkeit

Berechenbarkeit / Entscheidbarkeit

Wir interessieren uns für

- Entscheidungsprobleme, dargestellt als Mengen $P \subseteq \Sigma^*$

  Zum Beispiel das Leerheitsproblem für kontextfreie Grammatiken:

  Menge $\{\text{code}(G) \mid G \text{ kontextfreie Grammatik und } L(G) = \emptyset\}$

  wobei $\text{code}(G)$ die Kodierung der Grammatik G als Wort ist

- Funktionen $f : \mathbb{N}^k \to \mathbb{N}$ (Stelligkeit)

  Zum Beispiel:

  - Konstante Nullfunktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(x) = 0$
  - Binäre Additionsfunktion $f : \mathbb{N}^2 \to \mathbb{N}$ mit $f(x, y) = x + y$

Berechenbarkeit / Entscheidbarkeit

Intuitiv nennen wir

- Entscheidungsproblem $P \subseteq \Sigma^*$ entscheidbar wenn es einen Algorithmus gibt, der P entscheidet, also:

  bei Eingabe $w \in \Sigma^*$ Ausgabe 1 wenn $w \in P$ und Ausgabe 0 sonst

  (nach endlicher Zeit!)

- Funktion $f : \mathbb{N}^k \to \mathbb{N}$ berechenbar wenn es einen Algorithmus gibt, der f berechnet, also:

  bei Eingabe $(x_1, \ldots, x_k) \in \mathbb{N}^k$ Ausgabe $f(x_1, \ldots, x_n)$

  (nach endlicher Zeit!)

Wir werden diese Begriffe später formal definieren (Was ist “Algorithmus”?)

Berechenbarkeit / Entscheidbarkeit

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!

Berechenbarkeit / Entscheidbarkeit

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

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 $f(n) = 0$!)

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