

SS 2015

Einführung in die theoretische Informatik

Ernst W. Mayr

Fakultät für Informatik
TU München

<http://www14.in.tum.de/lehre/2015SS/theo/>

Sommersemester 2015

Kapitel 0 Organisatorisches

- Vorlesungen:
 - Mo 10:15–12:00, Do 16:00–17:45 MI HS1 F.L. Bauer (MI 00.02.001)
- Übung:
 - 2SWS Tutorübung: siehe Übungswebseite
Anmeldung in TUMonline
 - 2SWS Zentralübung (nicht verpflichtend): Do 14:30–15:55 MI HS1 F.L. Bauer (MI 00.02.001)
 - Übungsleitung: Dr. Werner Meixner
- Umfang:
 - 4V+2TÜ, 8 ECTS-Punkte
- Sprechstunde:
 - Mo 12:00–13:00 und nach Vereinbarung

- Übungsleitung:
 - Dr. W. Meixner, MI 03.09.040 (meixner@in.tum.de)
Sprechstunde: Do 12:00–12:30
- Sekretariat:
 - Frau Lissner, MI 03.09.052 (lissner@in.tum.de)

- Übungsaufgaben und Klausur:
 - Ausgabe jeweils am Montag auf der Webseite der Vorlesung
 - Abgabe Montag eine Woche später 13:00Uhr, Kästen bei den Schließfächern im Untergeschoß vor dem HS1
 - Besprechung in der Tutorübung
- Klausur:
 - Endklausur am **Donnerstag, 30. Juli 2015**, 11:00–14:00, Räume **MW2001 (5510.02.001, Rudolf-Diesel-Hörsaal)**, **MI HS1 (MI 00.02.001)**, **Interims-Hörsaal 1 (5620.01.101)**
 - Wiederholungsklausur am **Donnerstag, 24. September 2015**, 08:30–11:30, Raum tba
 - bei den Klausuren sind *keine* Hilfsmittel außer jeweils einem **eigenhändig** beschriebenen DIN-A4-Blatt zugelassen
 - Für das Bestehen des Moduls ist die erfolgreiche Teilnahme an der Abschlussklausur (mindestens 40% der Gesamtpunktzahl) **erforderlich**.
 - **Die Erfahrungen der letzten Jahre legen nahe, dass es für die erfolgreiche Bearbeitung der Abschlussklausur sehr förderlich ist, die angebotenen Hausaufgabenblätter zu bearbeiten (Sie erhalten sie korrigiert zurück), an der Tutorübung und auch(!) an der (freiwilligen) Zentralübung teilzunehmen!**
 - vorauss. 13 Übungsblätter, das letzte am 13. Juli 2015

- Vorkenntnisse:
 - Einführung in die Informatik 1/2
 - Diskrete Strukturen
 - Grundlagen: Algorithmen und Datenstrukturen
- Weiterführende Vorlesungen:
 - Effiziente Algorithmen und Datenstrukturen
 - Automaten, Formale Sprachen, Berechenbarkeit und Entscheidbarkeit
 - Logik
 - Komplexitätstheorie
 - Compilerbau
 - ...
- Webseite:

<http://www.mayr.in.tum.de/lehre/2015SS/theo/>

1. Ziel der Vorlesung

Der Zweck dieser Vorlesung ist das Studium fundamentaler Konzepte in der Theoretischen Informatik. Dies umfasst das Studium der Grundlagen formaler Sprachen und Automaten, von Berechnungsmodellen und Fragen der Entscheidbarkeit, die Diskussion algorithmischer Komplexität sowie einiger grundlegender Konzepte der Komplexitätstheorie.

Themengebiete werden also sein:

- Berechenbarkeitstheorie
 - Betrachtung und Untersuchung der Grenzen, was Rechner überhaupt können
- Komplexitätstheorie
 - Studium der Grenzen, was Rechner mit begrenzten Ressourcen leisten können
 - Herleitung *oberer* und *unterer* Schranken
- Automatentheorie
 - Rechner als endliche Systeme mit endlichem oder unendlichem Speicher
- Grammatiken
 - Aufbau von Programmiersprachen, Ausdruckskraft, Effizienz der Syntaxanalyse
- Algorithmen und ihre Komplexität

Historische Einordnung:

- 1936 Berechenbarkeitstheorie Church, Turing
- 1956 Automatentheorie, Reguläre Ausdrücke Kleene
- 1956 Grammatiken Chomsky
- 1971 Komplexitätstheorie Hennie, Stearns, Cook, Levin

2. Wesentliche Techniken und Konzepte

- Formalisierung und Abstraktion
 - Rechner werden durch mathematische Objekte nachgebildet
 - zu lösende Aufgaben werden mengentheoretisch als **Problem** definiert
 - die Abfolge von Berechnungsschritten wird formalisiert
 - die quantitative Bestimmung der Komplexität eines Verfahrens bzw. eines Problems wird festgelegt
- Simulation
 - Verfahren zur Ersetzung eines Programms in einem Formalismus A durch ein Programm in einem anderen Formalismus B bei unverändertem Ein-/Ausgabeverhalten

- Reduktion
 - formale Beschreibung für
“Problem A ist nicht (wesentlich) schwerer als Problem B”
- Äquivalenz
 - ein Formalismus A ist (prinzipiell) genauso mächtig wie Formalismus B
 - “Problem A und Problem B lassen sich (effizient) aufeinander reduzieren”
- Diagonalisierung
 - Auflistung aller Algorithmen einer bestimmten Klasse
 - Beweis durch Widerspruch
 - enger Bezug zu **Paradoxa**

3. Literatur




Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman:
The design and analysis of computer algorithms.
Addison-Wesley Publishing Company, Reading (MA), 1976





John E. Hopcroft, Jeffrey D. Ullman:
Introduction to automata theory, languages, and computation.
Addison-Wesley Publishing Company, Reading (MA), 1979



Uwe Schöning:
Theoretische Informatik — kurzgefasst.
Spektrum Akademischer Verlag GmbH, Heidelberg-Berlin, 1997

 Katrin Erk, Lutz Priese:
Theoretische Informatik: Eine umfassende Einführung (3. Auflage).
Springer-Verlag, Berlin-Heidelberg-New York, 2008
[http://link.springer.com.eaccess.ub.tum.de/book/10.1007%
2F978-3-540-76320-8](http://link.springer.com.eaccess.ub.tum.de/book/10.1007%2F978-3-540-76320-8)

 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest:
Introduction to algorithms.
McGraw-Hill Book Company, New York-St. Louis-San Francisco-Montreal-Toronto,
1990

 Thomas Ottmann, Peter Widmayer:
Algorithmen und Datenstrukturen.
B.I., Mannheim-Leipzig-Wien-Zürich, 1993
[http://link.springer.com.eaccess.ub.tum.de/book/10.1007%
2F978-3-8274-2804-2](http://link.springer.com.eaccess.ub.tum.de/book/10.1007%2F978-3-8274-2804-2)



Volker Heun:

Grundlegende Algorithmen.

Vieweg, 2000

<http://link.springer.com.eaccess.ub.tum.de/book/10.1007%2F978-3-322-80323-8>



Ingo Wegener:

Theoretische Informatik.

B.G. Teubner, Stuttgart, 1993

<http://link.springer.com.eaccess.ub.tum.de/book/10.1007%2F978-3-322-94004-9>

Kapitel I Formale Sprachen und Automaten

1. Beispiele

Sei Σ ein (endliches) Alphabet. Dann

Definition 1

- 1 ist ein **Wort**/String über Σ eine endliche Folge von Zeichen aus Σ ;
- 2 wird das leere Wort mit ϵ bezeichnet;
- 3 bezeichnet uv die **Konkatenation** der beiden Wörter u und v ;
- 4 ist Σ^* das **Monoid** über Σ , d.h. die Menge aller endlichen Wörter über Σ ;
- 5 ist Σ^+ die Menge aller nichtleeren endlichen Wörter über Σ ;
- 6 bezeichnet $|w|$ für $w \in \Sigma^*$ die Länge von w ;
- 7 ist, für jedes Wort w , w^n definiert durch $w^0 = \epsilon$ und $w^{n+1} = ww^n$;
- 8 ist Σ^n für $n \in \mathbb{N}_0$ die Menge aller Wörter der Länge n in Σ^* ;
- 9 eine Teilmenge $L \subseteq \Sigma^*$ eine **formale Sprache**.

Beispiel 2

Wir betrachten folgende Grammatik:

- ⟨Satz⟩ → ⟨Subjekt⟩⟨Prädikat⟩⟨Objekt⟩
- ⟨Subjekt⟩ → ⟨Artikel⟩⟨Attribut⟩⟨Substantiv⟩
- ⟨Artikel⟩ → ϵ
- ⟨Artikel⟩ → der|die|das|ein|...
- ⟨Attribut⟩ → ϵ |⟨Adjektiv⟩|⟨Adjektiv⟩⟨Attribut⟩
- ⟨Adjektiv⟩ → gross|klein|schön|...

Die vorletzte Ersetzungsregel ist **rekursiv**, die durch diese **Grammatik** definierte Sprache deshalb unendlich.

Beispiel 3 (Formale Sprachen)

- die Menge aller Wörter in der 24. Auflage des [Duden](#)
- $L_1 = \{aa, aaaa, aaaaaa, \dots\} = \{(aa)^n; n \in \mathbb{N}\}$ ($\Sigma_1 = \{a\}$)
- $L_2 = \{ab, abab, ababab, \dots\} = \{(ab)^n; n \in \mathbb{N}\}$ ($\Sigma_2 = \{a, b\}$)
- $L_3 = \{ab, aabb, aaabbb, \dots\} = \{a^n b^n; n \in \mathbb{N}\}$ ($\Sigma_3 = \{a, b\}$)
- $L_4 = \{a, b, aa, ab, bb, aaa, aab, abb, bbb, \dots\}$
 $= \{a^m b^n; m, n \in \mathbb{N}_0, m + n > 0\}$ ($\Sigma_4 = \{a, b\}$)
- $L_5 = \emptyset$
- $L_6 = \{\epsilon\}$
- $L_7 = \{x \in \Sigma^*; \text{ein gegebenes Programm/TM hält bei Eingabe } x\}$

Dagegen

Beispiel (Forts.)

- Die "Menge" der Sätze in deutscher Sprache ist *keine* formale Sprache
- ϵ ist *keine* formale Sprache
- \mathbb{R} ist *keine* formale Sprache

Definition 4 (Operationen auf Sprachen)

Seien $A, B \subseteq \Sigma^*$ zwei (formale) Sprachen.

- **Konkatenation:** $AB = \{uv; u \in A, v \in B\}$
- $A^0 = \{\epsilon\}$, $A^{n+1} = AA^n$
- $A^* = \bigcup_{n \geq 0} A^n$
- $A^+ = \bigcup_{n \geq 1} A^n$

Beispiel 5

- $\{ab, b\}\{a, bb\} = \{aba, abbb, ba, bbb\}$
 $\{ab, b\} \times \{a, bb\} = \{(ab, a), (ab, bb), (b, a), (b, bb)\}$
- $\{ab, b\}^2 = \{abab, abb, bab, bb\}$
- $\{ab, a\}\{ba, a\} = \{abba, aba, aa\}$
- $\emptyset^* = \{\epsilon\}$

Einige nützliche Rechenregeln:

- $\emptyset A = A\emptyset = \emptyset$
- $\{\epsilon\}A = A\{\epsilon\} = A$
- $A(B \cup C) = AB \cup AC$
- $(A \cup B)C = AC \cup BC$
- $A(B \cap C) = AB \cap AC$ gilt i.A. **nicht!**
- $A^*A^* = A^*$

Wie bekannt heißt eine Menge M **abzählbar**, falls sie gleich mächtig wie eine Teilmenge der natürlichen Zahlen \mathbb{N} (bzw. $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$) ist, d.h., falls eine Bijektion zwischen den beiden Mengen existiert.

Dies ist auch gleichbedeutend mit der Aussage, dass es eine Nummerierung der Elemente von M gibt, so dass

$$M = \{m_1, m_2, \dots\}.$$

Eine Menge heißt **überabzählbar**, falls sie nicht abzählbar ist.

Lemma 6

Für endliches Σ ist Σ^ abzählbar.*

Beweis:

Ohne Beweis. □

Bemerkungen:

- \mathbb{Q} ist abzählbar.
- \mathbb{R} und $[0, 1] \subset \mathbb{R}$ sind gleich mächtig (haben gleiche Kardinalität) und sind beide überabzählbar.

Satz 7

Die Menge der Sprachen über einem (nichtleeren) endlichen Alphabet ist überabzählbar.

Beweis:

Widerspruchsbeweis durch **Diagonalisierung**: Angenommen, L_0, L_1, L_2, \dots sei eine Nummerierung der Sprachen über Σ . Sei weiter w_0, w_1, w_2, \dots eine (feste) Abzählung von Σ^* .

Betrachte $L := \{w_i; w_i \notin L_i\}$. Dann kann L nicht in der Nummerierung L_0, L_1, L_2, \dots vorkommen! □

2. Die Chomsky-Hierarchie

Diese Sprachenhierarchie ist nach **Noam Chomsky** [MIT, 1976] benannt.

2.1 Phrasenstrukturgrammatik, Chomsky-Grammatik

Grammatiken bestehen aus

- 1 einem **Terminalalphabet** Σ (manchmal auch T), $|\Sigma| < \infty$
- 2 einem endlichen Vorrat von **Nichtterminalzeichen** (Variablen) V , $V \cap \Sigma = \emptyset$
- 3 einem **Startsymbol** (Axiom) $S \in V$
- 4 einer endliche Menge P von **Produktionen** (Ableitungsregeln) der Form $l \rightarrow r$, mit $l \in (V \cup \Sigma)^* V (V \cup \Sigma)^*$, $r \in (V \cup \Sigma)^*$

Eine **Phrasenstrukturgrammatik** (Grammatik) ist ein Quadrupel $G = (V, \Sigma, P, S)$.

Sei $G = (V, \Sigma, P, S)$ eine Phrasenstrukturgrammatik.

Definition 8

Wir schreiben

- 1 $z \rightarrow_G z'$ gdw $(\exists x, y \in (V \cup \Sigma)^*, l \rightarrow r \in P)[z = xly, z' = xry]$
- 2 $z \rightarrow_G^* z'$ gdw $z = z'$ oder $z \rightarrow_G z^{(1)} \rightarrow_G z^{(2)} \rightarrow_G \dots \rightarrow_G z^{(k)} = z'$. Eine solche Folge von Ableitungsschritten heißt eine **Ableitung für z' von z in G** (der Länge k).
- 3 Die von G **erzeugte Sprache** ist

$$L(G) := \{z \in \Sigma^*; S \rightarrow_G^* z\}$$

Zur Vereinfachung der Notation schreiben wir gewöhnlich \rightarrow und \rightarrow^* statt \rightarrow_G und \rightarrow_G^*

Vereinbarung:

Wir bezeichnen **Nichtterminale** mit großen und **Terminale** mit kleinen Buchstaben!

Beispiel 9

Wir erinnern uns:

- $L_2 = \{ab, abab, ababab, \dots\} = \{(ab)^n; n \in \mathbb{N}\}$ ($\Sigma_2 = \{a, b\}$)
- Grammatik für L_2 mit folgenden Produktionen:

$$S \rightarrow ab, S \rightarrow abS$$

Beispiel 9 (Forts.)

- $L_4 = \{a, b, aa, ab, bb, aaa, aab, abb, bbb \dots\}$
 $= \{a^m b^n; m, n \in \mathbb{N}_0, m + n > 0\}$ ($\Sigma_4 = \{a, b\}$)
- Grammatik für L_4 mit folgenden Produktionen:

$$\begin{aligned} S &\rightarrow A, S \rightarrow B, S \rightarrow AB, \\ A &\rightarrow a, A \rightarrow aA, \\ B &\rightarrow b, B \rightarrow bB \end{aligned}$$

2.2 Die Chomsky-Hierarchie

Sei $G = (V, \Sigma, P, S)$ eine Phrasenstrukturgrammatik.

- 1 Jede Phrasenstrukturgrammatik (Chomsky-Grammatik) ist (zunächst) automatisch vom **Typ 0**.
- 2 Eine Chomsky-Grammatik heißt (längen-)monoton, falls für alle Regeln

$$\alpha \rightarrow \beta \in P \text{ mit } \alpha \neq S$$

gilt:

$$|\alpha| \leq |\beta| ,$$

und, falls $S \rightarrow \epsilon \in P$, dann das Axiom S auf keiner rechten Seite vorkommt.

- ③ Eine Chomsky-Grammatik ist vom **Typ 1** (auch: **kontextsensitiv**), falls sie monoton ist und für alle Regeln $\alpha \rightarrow \beta$ in P mit $\alpha \neq S$ gilt:

$$\alpha = \alpha' A \alpha'' \text{ und } \beta = \alpha' \beta' \alpha''$$

für geeignete $A \in V$, $\alpha', \alpha'' \in (V \cup \Sigma)^*$ und $\beta' \in (V \cup \Sigma)^+$.

- ④ Eine Chomsky-Grammatik ist vom **Typ 2** (auch: **kontextfrei**), falls sie monoton ist und für alle Regeln $\alpha \rightarrow \beta \in P$ gilt:

$$\alpha \in V .$$

Bemerkung: Manchmal wird “kontextfrei” auch ohne die Monotonie-Bedingung definiert; **streng monoton** schließt dann die Monotonie mit ein, so dass ϵ nicht als rechte Seite vorkommen kann.

- 5 Eine Chomsky-Grammatik ist vom **Typ 3** (auch: **regulär**, **rechtslinear**), falls sie monoton ist und für alle Regeln $\alpha \rightarrow \beta$ in P mit $\beta \neq \epsilon$ gilt:

$$\alpha \in V \text{ und } \beta \in \Sigma^+ \cup \Sigma^*V .$$

Auch hier gilt die entsprechende Bemerkung zur Monotonie-Bedingung.

Beispiel 10

- Die folgende Grammatik ist regulär:

$$\begin{aligned} S &\rightarrow \epsilon, S \rightarrow A, \\ A &\rightarrow aa, A \rightarrow aaA \end{aligned}$$

- Eine Produktion

$$A \rightarrow Bcde$$

heißt **linkslinear**.

- Eine Produktion

$$A \rightarrow abcDef$$

heißt **linear**.

Definition 11

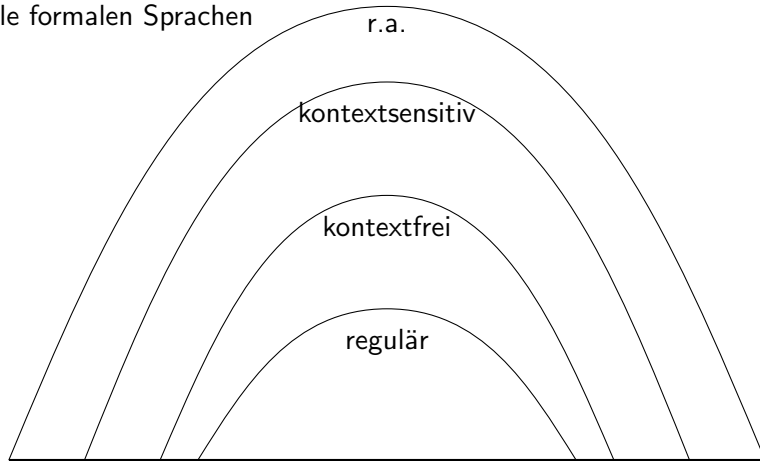
Eine Sprache $L \subseteq \Sigma^*$ heißt **vom Typ k** , $k \in \{0, 1, 2, 3\}$, falls es eine Chomsky- k -Grammatik G mit $L(G) = L$ gibt.

In der Chomsky-Hierarchie bilden also die Typ-3- oder regulären Sprachen die kleinste, unterste Stufe, darüber kommen die kontextfreien, dann die kontextsensitiven Sprachen. Oberhalb der Typ-1-Sprachen kommen die Typ-0-Sprachen, die auch **rekursiv aufzählbar** oder **semi-entscheidbar** genannt werden. Darüber (und nicht mehr Teil der Chomsky-Hierarchie) findet sich z.B. die Klasse aller formalen Sprachen.

In Typ-3-Grammatiken müssen entweder alle Produktionen rechtslinear oder alle linkslinear sein.

Überlegen Sie sich eine **lineare** Grammatik, deren Sprache nicht regulär ist!
(Beweismethode später!)

alle formalen Sprachen



Lemma 12

Sei $G = (V, \Sigma, P, S)$ eine Chomsky-Grammatik, so dass alle Produktionen $\alpha \rightarrow \beta$ die Bedingung $\alpha \in V$ erfüllen. Dann ist $L(G)$ kontextfrei.

Beweis:

Definition 13

Ein $A \in V$ mit $A \rightarrow^* \epsilon$
heißt **nullierbar**.

Bestimme alle nullierbaren $A \in V$:

```
 $N := \{A \in V; (A \rightarrow \epsilon) \in P\}$   
 $N' := \emptyset$   
while  $N \neq N'$  do  
   $N' := N$   
   $N := N' \cup \{A \in V;$   
     $(\exists (A \rightarrow \beta) \in P)[\beta \in N'^*]\}$   
od
```

Wie man leicht durch Induktion sieht, enthält N zum Schluss genau alle nullierbaren $A \in V$.

Sei nun G eine Grammatik, so dass alle linken Seiten $\in V$, aber die Monotoniebedingung nicht unbedingt erfüllt ist.

Modifiziere G zu G' mit Regelmenge P' wie folgt:

- 1 für jedes $(A \rightarrow x_1x_2 \cdots x_n) \in P$, $n \geq 1$, füge zu P' alle Regeln $A \rightarrow y_1y_2 \cdots y_n$ hinzu, die dadurch entstehen, dass für nicht-nullierbare x_i $y_i := x_i$ und für nullierbare x_i die beiden Möglichkeiten $y_i := x_i$ und $y_i := \epsilon$ eingesetzt werden, ohne dass die ganze rechte Seite $= \epsilon$ wird.
- 2 falls S nullierbar ist, sei T ein neues Nichtterminal; füge zu P' die Regeln $S \rightarrow \epsilon$ und $S \rightarrow T$ hinzu, ersetze S in allen rechten Seiten durch T und ersetze jede Regel $(S \rightarrow x) \in P'$, $|x| > 0$, durch $T \rightarrow x$.

Lemma 14

$G' = (V \cup T, \Sigma, P', S)$ ist kontextfrei, und es gilt

$$L(G') = L(G) .$$

Beweis:

Klar!



Auch für reguläre Grammatiken gilt ein entsprechender Satz über die “Entfernbarkeit” nullierbarer Nichtterminale:

Lemma 15

Sei $G = (V, \Sigma, P, S)$ eine Chomsky-Grammatik, so dass für alle Regeln $\alpha \rightarrow \beta \in P$ gilt:

$$\alpha \in V \text{ und } \beta \in \Sigma^* \cup \Sigma^*V .$$

Dann ist $L(G)$ regulär.

Beweis:

Übungsaufgabe!



Beispiel 16

Typ 3: $L = \{a^n; n \in \mathbb{N}\}$, Grammatik: $S \rightarrow a,$
 $S \rightarrow aS$

Typ 2: $L = \{a^n b^n; n \in \mathbb{N}_0\}$, Grammatik: $S \rightarrow \epsilon,$
 $S \rightarrow T,$
 $T \rightarrow ab,$
 $T \rightarrow aTb$

Wir benötigen beim Scannen *einen* Zähler.

Beispiel 16 (Forts.)

Typ 1: $L = \{a^n b^n c^n; n \in \mathbb{N}\}$, Grammatik:

$$\begin{aligned} S &\rightarrow aSXY, \\ S &\rightarrow abY, \\ YX &\rightarrow XY, \\ bX &\rightarrow bb, \\ bY &\rightarrow bc, \\ cY &\rightarrow cc \end{aligned}$$

Wir benötigen beim Scannen *mindestens zwei* Zähler.

Bemerkung: Diese Grammatik entspricht *nicht* unserer Definition des Typs 1, sie ist aber (längen-)monoton. Wir zeigen als Hausaufgabe, dass monotone und Typ 1 Grammatiken die gleiche Sprachklasse erzeugen!

Die **Backus-Naur-Form** (BNF) ist ein Formalismus zur kompakten Darstellung von Typ-2-Grammatiken.

- Statt

$$\begin{aligned} A &\rightarrow \beta_1 \\ A &\rightarrow \beta_2 \\ &\vdots \\ A &\rightarrow \beta_n \end{aligned}$$

schreibt man

$$A \rightarrow \beta_1 | \beta_2 | \dots | \beta_n .$$

Die **Backus-Naur-Form** (BNF) ist ein Formalismus zur kompakten Darstellung von Typ-2-Grammatiken.

- Statt

$$A \rightarrow \alpha\gamma$$

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

schreibt man

$$A \rightarrow \alpha[\beta]\gamma.$$

(D.h., das Wort β kann, muss aber nicht, zwischen α und γ eingefügt werden.)

Die **Backus-Naur-Form** (BNF) ist ein Formalismus zur kompakten Darstellung von Typ-2-Grammatiken.

- Statt

$$A \rightarrow \alpha\gamma$$

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

$$B \rightarrow \beta$$

$$B \rightarrow \beta B$$

schreibt man

$$A \rightarrow \alpha\{\beta\}\gamma.$$

(D.h., das Wort β kann beliebig oft (auch Null mal) zwischen α und γ eingefügt werden.)

Beispiel 17

⟨Satz⟩ → ⟨Subjekt⟩⟨Prädikat⟩⟨Objekt⟩
⟨Subjekt⟩ → ⟨Artikel⟩⟨Attribut⟩⟨Substantiv⟩
⟨Prädikat⟩ → ist|hat|...
⟨Artikel⟩ → ε|der|die|das|ein|...
⟨Attribut⟩ → {⟨Adjektiv⟩}
⟨Adjektiv⟩ → gross|klein|schön|...
⟨Substantiv⟩ → ...

2.3 Das Wortproblem

Beispiel 18 (Arithmetische Ausdrücke)

$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{term} \rangle \\ \langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \\ \langle \text{term} \rangle &\rightarrow (\langle \text{expr} \rangle) \\ \langle \text{term} \rangle &\rightarrow \langle \text{term} \rangle \times \langle \text{term} \rangle \\ \langle \text{term} \rangle &\rightarrow a \mid b \mid \dots \mid z\end{aligned}$$

Aufgabe eines **Parsers** ist nun, zu prüfen, ob eine gegebene Zeichenreihe einen gültigen arithmetischen Ausdruck darstellt und, falls ja, ihn in seine Bestandteile zu zerlegen.

Sei $G = (V, \Sigma, P, S)$ eine Grammatik.

Definition 19

- ① **Wortproblem:** Gegeben ein Wort $w \in \Sigma^*$, stelle fest, ob

$$w \in L(G) ?$$

- ② **Ableitungsproblem:** Gegeben ein Wort $w \in L(G)$, gib eine Ableitung $S \rightarrow_G^* w$ an, d.h. eine Folge

$$S = w^{(0)} \rightarrow_G w^{(1)} \rightarrow_G \cdots \rightarrow_G w^{(n)} = w$$

mit $w^{(i)} \in (\Sigma \cup V)^*$ für $i = 1, \dots, n$.

- ③ **uniformes Wortproblem:** Wortproblem, bei dem jede Probleminstance sowohl die Grammatik G wie auch die zu testende Zeichenreihe w enthält. Ist G dagegen **global** festgelegt, spricht man von einem **nicht-uniformen** Wortproblem.

Bemerkung:

Das uniforme wie auch das nicht-uniforme Wortproblem ist für Typ-0-Sprachen (also die rekursiv-aufzählbare Sprachen) im Allgemeinen nicht entscheidbar. Wir werden später sehen, dass es zum [Halteproblem für Turingmaschinen](#) äquivalent ist.

Es gilt jedoch

Satz 20

Für kontextsensitive Grammatiken ist das Wortproblem entscheidbar.

Genauer: Es gibt einen Algorithmus, der bei Eingabe einer kontextsensitiven Grammatik $G = (V, \Sigma, P, S)$ und eines Wortes w in endlicher Zeit entscheidet, ob $w \in L(G)$.

Beweisidee:

Angenommen $w \in L(G)$. Dann gibt es eine Ableitung

$$S = w^{(0)} \rightarrow_G w^{(1)} \rightarrow_G \cdots \rightarrow_G w^{(\ell)} = w$$

mit $w^{(i)} \in (\Sigma \cup V)^*$ für $i = 1, \dots, \ell$.

Da aber G kontextsensitiv ist, gilt (falls $w \neq \epsilon$)

$$|w^{(0)}| \leq |w^{(1)}| \leq \cdots \leq |w^{(\ell)}| ,$$

d.h., es genügt, nur Wörter in $(\Sigma \cup V)^*$ der Länge $\leq |w|$ zu betrachten.

Beweis:

Sei o.B.d.A. $w \neq \epsilon$ und sei $T_m^n := \{w' \in (\Sigma \cup V)^*; |w'| \leq n \text{ und } w' \text{ lässt sich aus } S \text{ in } \leq m \text{ Schritten ableiten}\}$

Diese Mengen kann man für alle n und m induktiv wie folgt berechnen:

$$\begin{aligned} T_0^n &:= \{S\} \\ T_{m+1}^n &:= T_m^n \cup \{w' \in (\Sigma \cup V)^*; |w'| \leq n \text{ und } w'' \rightarrow w' \text{ für ein } w'' \in T_m^n\} \end{aligned}$$

Beachte: Für alle m gilt: $|T_m^n| \leq \sum_{i=1}^n |\Sigma \cup V|^i$.

Es muss daher, für festes n , immer ein m_0 geben mit

$$T_{m_0}^n = T_{m_0+1}^n = \dots$$

Beweis (Forts.):

Algorithmus:

$n := |w|$

$T := \{S\}$

$T' := \emptyset$

while $T \neq T'$ **do**

$T' := T$

$T := T' \cup \{w' \in (V \cup \Sigma)^+; |w'| \leq n, (\exists w'' \in T')[w'' \rightarrow w']\}$

od

if $w \in T$ **return** „ja“ **else return** „nein“ **fi**

□

Beispiel 21

Gegeben sei die Typ-2-Grammatik mit den Produktionen

$$S \rightarrow ab \text{ und } S \rightarrow aSb$$

sowie das Wort $w = abab$.

$$T_0^4 = \{S\}$$

$$T_1^4 = \{S, ab, aSb\}$$

$$T_2^4 = \{S, ab, aSb, aabb\} \quad aaSbb \text{ ist zu lang!}$$

$$T_3^4 = \{S, ab, aSb, aabb\}$$

Also lässt sich das Wort w mit der gegebenen Grammatik **nicht** erzeugen!

Bemerkung:

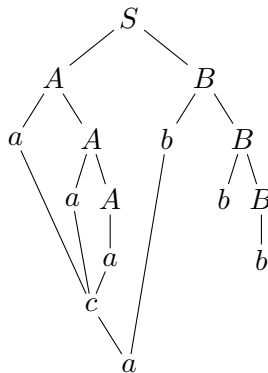
Der angegebene Algorithmus ist nicht sehr effizient! Für **kontextfreie** Grammatiken gibt es wesentlich effizientere Verfahren, die wir später kennenlernen werden!

2.4 Ableitungsgraph und Ableitungsbaum

Grammatik:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \\ A &\rightarrow a \\ B &\rightarrow bB \\ B &\rightarrow b \\ aaa &\rightarrow c \\ cb &\rightarrow a \end{aligned}$$

Beispiel:

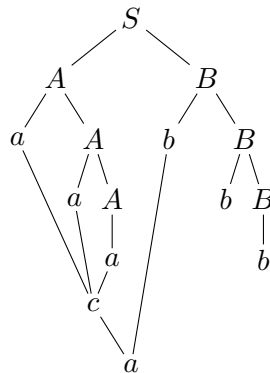


Die Symbole ohne Kante nach unten entsprechen, von links nach rechts gelesen, dem durch den Ableitungsgraphen dargestellten Wort.

Grammatik:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \\ A &\rightarrow a \\ B &\rightarrow bB \\ B &\rightarrow b \\ aaa &\rightarrow c \\ cb &\rightarrow a \end{aligned}$$

Beispiel:



Dem Ableitungsgraph entspricht z.B. die Ableitung

$$\begin{aligned} S &\rightarrow AB \rightarrow aAB \rightarrow aAbB \rightarrow aaAbB \rightarrow aaAbbB \rightarrow \\ &\rightarrow aaabbB \rightarrow aaabbb \rightarrow cbbb \rightarrow abb \end{aligned}$$

Beobachtung:

Bei kontextfreien Sprachen sind die Ableitungsgraphen immer Bäume.

Beispiel 22

Grammatik:

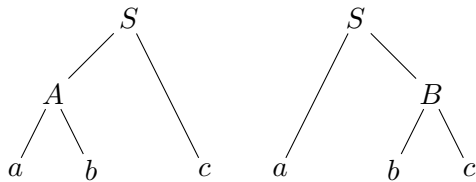
$$S \rightarrow aB$$

$$S \rightarrow Ac$$

$$A \rightarrow ab$$

$$B \rightarrow bc$$

AbleitungsbaumAbleitungsbäume:



Für das Wort abc gibt es zwei verschiedene Ableitungsbäume.

Definition 23

- Eine Ableitung

$$S = w^{(0)} \rightarrow w^{(1)} \rightarrow \dots \rightarrow w^{(n)} = w$$

eines Wortes w heißt **Linksableitung**, wenn für jede Anwendung einer Produktion $\alpha \rightarrow \beta$ auf $w^{(i)} = x\alpha z$ gilt, dass sich **keine** Regel der Grammatik auf ein echtes Präfix von $x\alpha$ anwenden lässt.

- Eine Grammatik heißt **eindeutig**, wenn es für jedes Wort $w \in L(G)$ genau eine Linksableitung gibt. Nicht eindeutige Grammatiken nennt man auch **mehrdeutig**.
- Eine Sprache L heißt **eindeutig**, wenn es für L eine eindeutige Grammatik gibt. Ansonsten heißt L mehrdeutig.

Bemerkung: Eindeutigkeit wird meist für kontextfreie (und reguläre) Grammatiken betrachtet, ist aber allgemeiner definiert.

Beispiel 24

Grammatik:

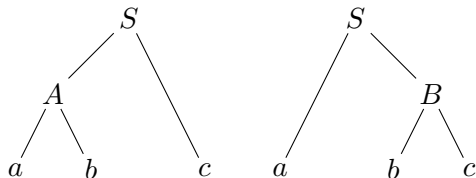
$$S \rightarrow aB$$

$$S \rightarrow Ac$$

$$A \rightarrow ab$$

$$B \rightarrow bc$$

Ableitungsbäume:



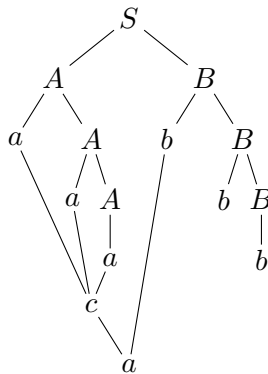
Beide Ableitungsbäume für das Wort abc entsprechen Linksableitungen.

Beispiel 25

Grammatik:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \\ A &\rightarrow a \\ B &\rightarrow bB \\ B &\rightarrow b \\ aaa &\rightarrow c \\ cb &\rightarrow a \end{aligned}$$

Ableitung:



Eine Linksableitung ist

$$\begin{aligned} S &\rightarrow AB \rightarrow aAB \rightarrow aaAB \rightarrow aaaB \rightarrow cB \rightarrow \\ &\rightarrow cbB \rightarrow aB \rightarrow abB \rightarrow abb \end{aligned}$$

Beispiel 25

Grammatik:

$$S \rightarrow AB$$

$$A \rightarrow aA$$

$$A \rightarrow a$$

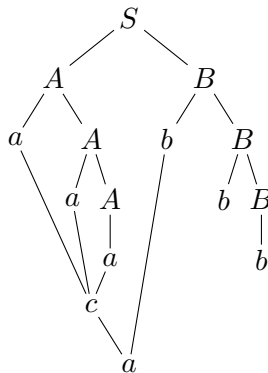
$$B \rightarrow bB$$

$$B \rightarrow b$$

$$aaa \rightarrow c$$

$$cb \rightarrow a$$

Ableitung:



Eine andere Linksableitung für abb ist

$$S \rightarrow AB \rightarrow aB \rightarrow abB \rightarrow abb .$$

Beispiel 25

Grammatik:

$$S \rightarrow AB$$

$$A \rightarrow aA$$

$$A \rightarrow a$$

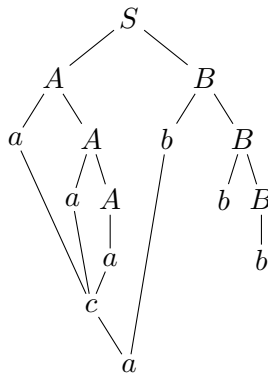
$$B \rightarrow bB$$

$$B \rightarrow b$$

$$aaa \rightarrow c$$

$$cb \rightarrow a$$

Ableitung:



Die Grammatik ist also **mehrdeutig**.

3. Reguläre Sprachen

3.1 Deterministische endliche Automaten

Definition 26

Ein **deterministischer endlicher Automat** (englisch: deterministic finite automaton, kurz DFA) wird durch ein 5-Tupel $M = (Q, \Sigma, \delta, q_0, F)$ beschrieben, das folgende Bedingungen erfüllt:

- 1 Q ist eine endliche Menge von **Zuständen**.
- 2 Σ ist eine endliche Menge, das **Eingabealphabet**, wobei $Q \cap \Sigma = \emptyset$.
- 3 $q_0 \in Q$ ist der **Startzustand**.
- 4 $F \subseteq Q$ ist die Menge der **Endzustände** (oder auch **akzeptierenden Zustände**)
- 5 $\delta : Q \times \Sigma \rightarrow Q$ heißt **Übergangsfunktion**.

Die von M akzeptierte/erkannte Sprache ist

$$L(M) := \{w \in \Sigma^*; \hat{\delta}(q_0, w) \in F\},$$

wobei $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ induktiv definiert ist durch

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= q && \text{für alle } q \in Q \\ \hat{\delta}(q, ax) &= \hat{\delta}(\delta(q, a), x) && \text{für alle } q \in Q, a \in \Sigma \\ &&& \text{und } x \in \Sigma^* \end{aligned}$$

Bemerkung: Endliche Automaten können durch (gerichtete und markierte) Zustandsgraphen veranschaulicht werden:

- Knoten $\hat{=}$ Zuständen
- Kanten $\hat{=}$ Übergängen
- genauer: eine mit $a \in \Sigma$ markierte Kante (u, v) entspricht $\delta(u, a) = v$

Der Anfangszustand wird durch einen Pfeil, Endzustände werden durch doppelte Kreise gekennzeichnet.

Beispiel 27

Sei $M = (Q, \Sigma, \delta, q_0, F)$,

wobei

$$Q = \{q_0, q_1, q_2, q_3\}$$

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

$$F = \{q_3\}$$

$$\delta(q_0, a) = q_1$$

$$\delta(q_0, b) = q_3$$

$$\delta(q_1, a) = q_2$$

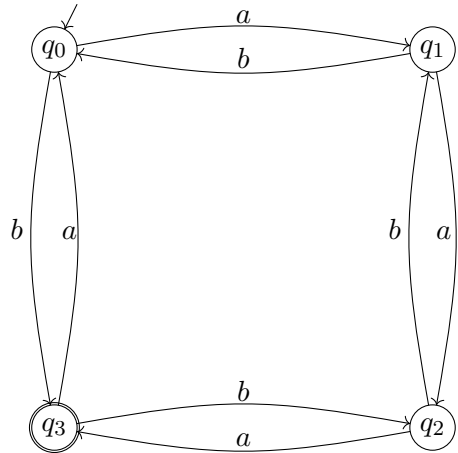
$$\delta(q_1, b) = q_0$$

$$\delta(q_2, a) = q_3$$

$$\delta(q_2, b) = q_1$$

$$\delta(q_3, a) = q_0$$

$$\delta(q_3, b) = q_2$$



Satz 28

Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer endlicher Automat und

$$P := \{q \rightarrow aq'; \delta(q, a) = q'\} \cup \{q \rightarrow a; \delta(q, a) \in F\}$$

eine Menge von Produktionen, zu der wir, falls $q_0 \in F$, noch die Produktion $q_0 \rightarrow \epsilon$ hinzufügen (und dann, falls nötig, nämlich wenn q_0 auf der rechten Seite einer Produktion vorkommt, die Monotoniebedingung wiederherstellen).

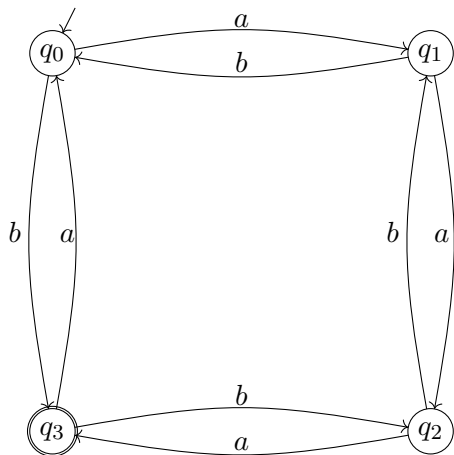
Dann ist die Grammatik $G = (Q, \Sigma, P, q_0)$ regulär.

Beweis:

Offensichtlich!



Beispiel 29



Produktionen:

q_0	\rightarrow	aq_1	q_0	\rightarrow	bq_3
q_1	\rightarrow	aq_2	q_1	\rightarrow	bq_0
q_2	\rightarrow	aq_3	q_2	\rightarrow	bq_1
q_3	\rightarrow	aq_0	q_3	\rightarrow	bq_2
q_2	\rightarrow	a	q_0	\rightarrow	b

$q_0 \rightarrow aq_1 \rightarrow abq_0$
 $\rightarrow abaq_1 \rightarrow abaaq_2$
 $\rightarrow abaaa \in L(G)$

Satz 30

Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein endlicher deterministischer Automat. Dann gilt für die soeben konstruierte reguläre Grammatik G

$$L(G) = L(M) .$$

Beweis:

Der Fall $w = \epsilon$ ist klar. Sei nun $w = a_1 a_2 \cdots a_n \in \Sigma^+$. Dann gilt gemäß Konstruktion:

$$w \in L(M)$$

$$\Leftrightarrow \exists q_0, q_1, \dots, q_n \in Q: q_0 \text{ Startzustand von } M,$$

$$\forall i = 0, \dots, n-1: \delta(q_i, a_{i+1}) = q_{i+1}, q_n \in F$$

$$\Leftrightarrow \exists q_0, q_1, \dots, q_{n-1} \in V: q_0 \text{ Startsymbol von } G$$

$$q_0 \rightarrow a_1 q_1 \rightarrow a_1 a_2 q_2 \rightarrow \cdots \rightarrow a_1 \cdots a_{n-1} q_{n-1} \rightarrow$$

$$\rightarrow a_1 \cdots a_{n-1} a_n$$

$$\Leftrightarrow w \in L(G)$$



3.2 Nichtdeterministische endliche Automaten

Definition 31

Ein **nichtdeterministischer endlicher Automat** (englisch: nondeterministic finite automaton, kurz NFA) wird durch ein 5-Tupel $N = (Q, \Sigma, \delta, S, F)$ beschrieben, das folgende Bedingungen erfüllt:

- 1 Q ist eine endliche Menge von **Zuständen**.
- 2 Σ ist eine endliche Menge, das **Eingabealphabet**, wobei $Q \cap \Sigma = \emptyset$.
- 3 $S \subseteq Q$ ist die Menge der **Startzustände**.
- 4 $F \subseteq Q$ ist die Menge der **Endzustände**.
- 5 $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ heißt **Übergangsrelation**.

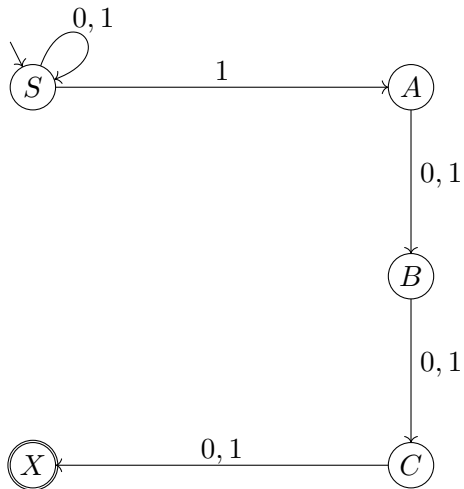
Die von N akzeptierte Sprache ist

$$L(N) := \{w \in \Sigma^*; \hat{\delta}(S, w) \cap F \neq \emptyset\},$$

wobei $\hat{\delta} : \mathcal{P}(Q) \times \Sigma^* \rightarrow \mathcal{P}(Q)$ wieder induktiv definiert ist durch

$$\begin{aligned}\hat{\delta}(Q', \epsilon) &= Q' \quad \forall Q' \subseteq Q \\ \hat{\delta}(Q', ax) &= \hat{\delta}\left(\bigcup_{q \in Q'} \delta(q, a), x\right) \quad \forall Q' \subseteq Q, \forall a \in \Sigma, \forall x \in \Sigma^*\end{aligned}$$

Beispiel 32



NFA für Binärzeichenreihen, deren viertletzttes Zeichen 1 ist

3.3 Äquivalenz von NFA und DFA

Satz 33

Für jede von einem nichtdeterministischen endlichen Automaten akzeptierte Sprache L gibt es auch einen deterministischen endlichen Automaten M mit

$$L = L(M) .$$

Beweis:

Sei $N = (Q, \Sigma, \delta, S, F)$ ein NFA.

Definiere

- 1 $M' := (Q', \Sigma, \delta', q'_0, F')$, mit
- 2 $Q' := \mathcal{P}(Q)$ ($\mathcal{P}(Q) = 2^Q$ Potenzmenge von Q)
- 3 $\delta'(Q'', a) := \bigcup_{q' \in Q''} \delta(q', a)$ für alle $Q'' \in Q'$, $a \in \Sigma$
- 4 $q'_0 := S$
- 5 $F' := \{Q'' \subseteq Q; Q'' \cap F \neq \emptyset\}$

Also

NFA N :	Q	Σ	δ	S	F
DFA M' :	2^Q	Σ	δ'	S	F'

Beweis (Forts.):

Es gilt:

$$\begin{aligned}w \in L(N) &\Leftrightarrow \hat{\delta}(S, w) \cap F \neq \emptyset \\ &\Leftrightarrow \hat{\delta}'(q'_0, w) \in F' \\ &\Leftrightarrow w \in L(M').\end{aligned}$$



Der zugehörige Algorithmus zur Überführung eines NFA in einen DFA heißt **Teilmengenkonstruktion**, **Potenzmengenkonstruktion** oder **Myhill-Konstruktion**.

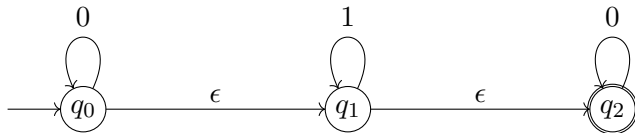
3.4 NFA's mit ϵ -Übergängen

Definition 34

Ein (nichtdeterministischer) endlicher Automat A mit ϵ -Übergängen ist ein 5-Tupel analog zur Definition des NFA mit

$$\delta : Q \times (\Sigma \uplus \{\epsilon\}) \rightarrow \mathcal{P}(Q) .$$

Ein ϵ -Übergang wird ausgeführt, ohne dass ein Eingabezeichen gelesen wird. Wir setzen o.B.d.A. voraus, dass A nur einen Anfangszustand hat.



Definiere für alle $a \in \Sigma$

$$\bar{\delta}(q, a) := \hat{\delta}(q, \epsilon^* a \epsilon^*).$$

Falls A das leere Wort ϵ mittels ϵ -Übergängen akzeptiert, also $F \cap \hat{\delta}(q_0, \epsilon^*) \neq \emptyset$, dann setze zusätzlich

$$F := F \cup \{q_0\}.$$

Satz 35

$$w \in L(A) \Leftrightarrow \hat{\delta}(S, w) \cap F \neq \emptyset.$$

Beweis:

Hausaufgabe!



3.5 Entfernen von ϵ -Übergängen

Satz 36

Zu jedem nichtdeterministischen endlichen Automaten A mit ϵ -Übergängen gibt es einen nichtdeterministischen endlichen Automaten A' ohne ϵ -Übergänge, so dass gilt:

$$L(A) = L(A')$$

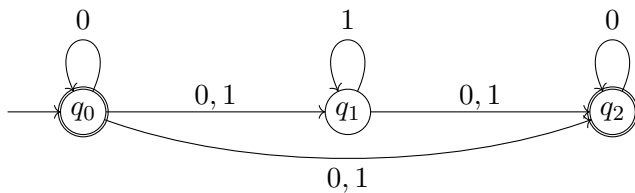
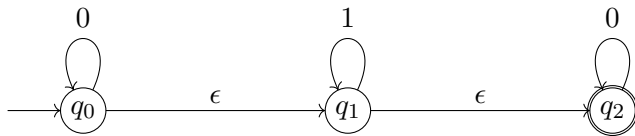
Beweis:

Ersetze δ durch $\bar{\delta}$ und F durch F' mit

$$F' = \begin{cases} F & \epsilon \notin L(A) \\ F \cup \{q_0\} & \epsilon \in L(A) \end{cases}$$



Beispiel 37



3.6 Endliche Automaten und reguläre Sprachen

Satz 38

Ist $G = (V, \Sigma, P, S)$ eine rechtslineare (also reguläre) Grammatik (o.B.d.A. sind die rechten Seiten aller Produktionen aus $\Sigma \cup \Sigma V$), so ist $N = (V \uplus \{X\}, \Sigma, \delta, \{S\}, F)$, (wobei X ein neues Nichtterminal-Symbol ist), mit

$$F := \begin{cases} \{S, X\}, & \text{falls } S \rightarrow \epsilon \in P \\ \{X\}, & \text{sonst} \end{cases}$$

und, für alle $A, B \in V, a \in \Sigma \cup \{\epsilon\}$,

$$\begin{aligned} B \in \delta(A, a) &\iff A \rightarrow aB && \text{und} \\ X \in \delta(A, a) &\iff A \rightarrow a \end{aligned}$$

ein nichtdeterministischer endlicher Automat, der genau $L(G)$ akzeptiert.

Beweis:

Aus der Konstruktion folgt, dass N ein NFA ist (i.A. mit ϵ -Übergängen).

Durch eine einfache Induktion über n zeigt man, dass eine Satzform

$$a_1 a_2 \cdots a_{n-1} A \text{ bzw. } a_1 a_2 \cdots a_n$$

in G genau dann ableitbar ist, wenn für die erweiterte Übergangsfunktion $\hat{\delta}$ des zu N äquivalenten NFA *ohne* ϵ -Übergänge gilt:

$$A \in \hat{\delta}(S, a_1 a_2 \cdots a_{n-1})$$

bzw.

$$X \in \hat{\delta}(S, a_1 a_2 \cdots a_n)$$

(bzw., für $n = 0$, $F \cap \hat{\delta}(S, \epsilon) \neq \emptyset$).



Zusammenfassend ergibt sich:

Satz 39

Die Klasse der regulären Sprachen (Chomsky-3-Sprachen) ist identisch mit der Klasse der Sprachen, die

- *von DFA's akzeptiert/erkannt werden,*
- *von NFA's akzeptiert werden,*
- *von NFA's mit ϵ -Übergängen akzeptiert werden.*

Beweis:

Wie soeben gezeigt.



3.7 Reguläre Ausdrücke

Reguläre Ausdrücke sollen eine kompakte Notation für spezielle Sprachen sein, wobei endliche Ausdrücke hier auch unendliche Mengen beschreiben können.

Definition 40

Reguläre Ausdrücke sind induktiv definiert durch:

- 1 \emptyset ist ein regulärer Ausdruck.
- 2 ϵ ist ein regulärer Ausdruck.
- 3 Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck.
- 4 Wenn α und β reguläre Ausdrücke sind, dann sind auch (α) , $\alpha\beta$, $(\alpha|\beta)$ (hierfür wird oft auch $(\alpha + \beta)$ geschrieben) und $(\alpha)^*$ reguläre Ausdrücke.
- 5 Nichts sonst ist ein regulärer Ausdruck.

Bemerkung: Ist α atomar, so schreiben wir statt $(\alpha)^*$ oft auch nur α^* .

Zu einem regulären Ausdruck γ ist die zugehörige Sprache $L(\gamma)$ induktiv definiert durch:

Definition 41

- 1 Falls $\gamma = \emptyset$, so gilt $L(\gamma) = \emptyset$.
- 2 Falls $\gamma = \epsilon$, so gilt $L(\gamma) = \{\epsilon\}$.
- 3 Falls $\gamma = a$, so gilt $L(\gamma) = \{a\}$.
- 4 Falls $\gamma = (\alpha)$, so gilt $L(\gamma) = L(\alpha)$.

- 5 Falls $\gamma = \alpha\beta$, so gilt

$$L(\gamma) = L(\alpha)L(\beta) = \{uv; u \in L(\alpha), v \in L(\beta)\} .$$

- 6 Falls $\gamma = (\alpha \mid \beta)$, so gilt

$$L(\gamma) = L(\alpha) \cup L(\beta) = \{u; u \in L(\alpha) \vee u \in L(\beta)\} .$$

- 7 Falls $\gamma = (\alpha)^*$, so gilt

$$L(\gamma) = L(\alpha)^* = \{u_1u_2 \dots u_n; n \in \mathbb{N}_0, u_1, \dots, u_n \in L(\alpha)\} .$$

Beispiel 42

Sei das zugrunde liegende Alphabet $\Sigma = \{0, 1\}$.

- alle Wörter, die gleich 0 sind oder mit 00 enden:

$$(0 \mid (0 \mid 1)^*00)$$

- alle Wörter, die 0110 enthalten:

$$(0|1)^*0110(0|1)^*$$

- alle Wörter, die eine gerade Anzahl von 1'en enthalten:

$$(0^*10^*1)^*0^*$$

- alle Wörter, die die Binärdarstellung einer durch 3 teilbaren Zahl darstellen, also

0, 11, 110, 1001, 1100, 1111, 10010, ...

Hausaufgabe!

Satz 43

Eine Sprache $L \subseteq \Sigma^*$ ist genau dann durch einen regulären Ausdruck darstellbar, wenn sie regulär ist.

Beweis:

“ \implies ”:

Sei also $L = L(\gamma)$.

Wir zeigen: \exists NFA N mit $L = L(N)$ mit Hilfe **struktureller** Induktion.

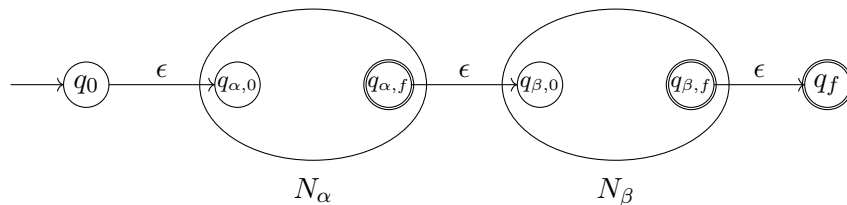
Induktionsanfang: Falls $\gamma = \emptyset$, $\gamma = \epsilon$, oder $\gamma = a \in \Sigma$, so folgt die Behauptung unmittelbar.

Induktionsschritt:

$$\gamma = \alpha\beta:$$

nach Induktionsannahme \exists NFA N_α und N_β mit

$$L(N_\alpha) = L(\alpha) \text{ und } L(N_\beta) = L(\beta) .$$

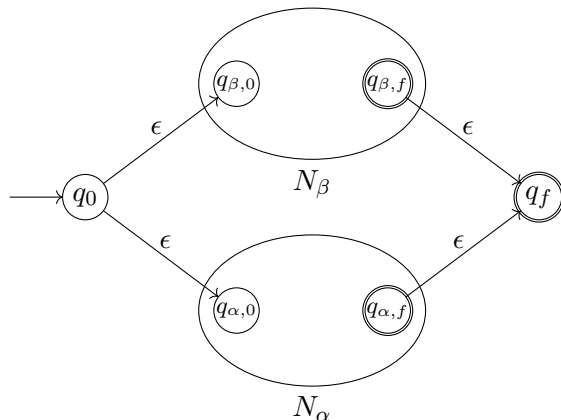


Induktionsschritt (Forts.):

$\gamma = (\alpha \mid \beta)$:

nach Induktionsannahme \exists NFA N_α und N_β mit

$$L(N_\alpha) = L(\alpha) \text{ und } L(N_\beta) = L(\beta) .$$

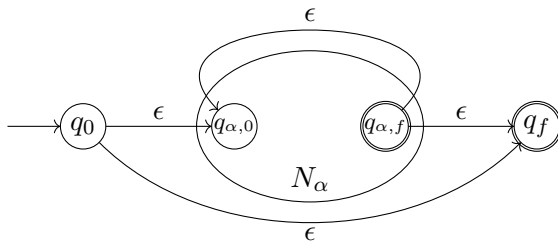


Induktionsschritt (Forts.):

$\gamma = (\alpha)^*$:

nach Induktionsannahme \exists NFA N_α mit

$$L(N_\alpha) = L(\alpha) .$$



“ \Leftarrow ”:

Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer endlicher Automat. Wir zeigen: es gibt einen regulären Ausdruck γ mit $L(M) = L(\gamma)$.

Sei $Q = \{q_0, \dots, q_n\}$. Wir setzen

$R_{ij}^k := \{w \in \Sigma^*; \text{ die Eingabe } w \text{ überführt den im Zustand } q_i \text{ gestarteten Automaten in den Zustand } q_j, \text{ wobei alle zwischendurch durchlaufenen Zustände einen Index kleiner gleich } k \text{ haben}\}$

Behauptung: Für alle $i, j \in \{0, \dots, n\}$ und alle $k \in \{-1, 0, 1, \dots, n\}$ gilt: Es gibt einen regulären Ausdruck α_{ij}^k mit $L(\alpha_{ij}^k) = R_{ij}^k$.

Bew.:

Induktion über k :

$k = -1$: Hier gilt

$$R_{ij}^{-1} := \begin{cases} \{a \in \Sigma; \delta(q_i, a) = q_j\}, & \text{falls } i \neq j \\ \{a \in \Sigma; \delta(q_i, a) = q_j\} \cup \{\epsilon\}, & \text{falls } i = j \end{cases}$$

R_{ij}^{-1} ist also endlich und lässt sich daher durch einen regulären Ausdruck α_{ij}^{-1} beschreiben.

Bew.:

Induktion über k :

$k \Rightarrow k + 1$: Hier gilt

$$R_{ij}^{k+1} = R_{ij}^k \cup R_{i k+1}^k (R_{k+1 k+1}^k)^* R_{k+1 j}^k$$
$$\alpha_{ij}^{k+1} = (\alpha_{ij}^k \mid \alpha_{i k+1}^k (\alpha_{k+1 k+1}^k)^* \alpha_{k+1 j}^k)$$

Somit gilt: $L(M) = L((\alpha_{0 f_1}^n \mid \alpha_{0 f_2}^n \mid \cdots \mid \alpha_{0 f_r}^n))$, wobei f_1, \dots, f_r die Indizes der Endzustände seien.

□(Satz 43)

3.8 Abschlusseigenschaften regulärer Sprachen

Satz 44

Seien $R_1, R_2 \subseteq \Sigma^*$ reguläre Sprachen. Dann sind auch

$$R_1 R_2, R_1 \cup R_2, R_1^*, \Sigma^* \setminus R_1 (=:\bar{R}_1), R_1 \cap R_2$$

reguläre Sprachen.

Beweis:

$R_1 R_2, R_1 \cup R_2, R_1^*$ klar.

$\Sigma^* \setminus R_1$: Sei $R_1 = L(A)$, A DFA, $A = (Q, \Sigma, \delta, q_0, F)$,
 δ vollständig.

Betrachte $A' = (Q, \Sigma, \delta, q_0, Q \setminus F)$.

Dann ist $L(A') = \Sigma^* \setminus L(A)$

$R_1 \cap R_2$: De Morgan



Die Produktkonstruktion für DFAs

Zwei DFAs laufen parallel und synchron, ein Eingabewort wird akzeptiert gdw beide Automaten es akzeptieren.

Satz 45

Seien $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ und $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ zwei DFAs. Dann ist der Produkt-Automat

$$M := (Q_1 \times Q_2, \Sigma, \delta, (s_1, s_2), F_1 \times F_2)$$

mit $\delta((q_1, q_2), a) := (\delta_1(q_1, a), \delta_2(q_2, a))$ für alle $q_1 \in Q_1, q_2 \in Q_2$ und $a \in \Sigma$ ein DFA, der $L(M_1) \cap L(M_2)$ erkennt.

Beweis:

Induktion über $|w|$. Es gilt:

$$\begin{aligned}w \in L(M) &\Leftrightarrow \hat{\delta}((s_1, s_2), w) \in F_1 \times F_2 \\&\Leftrightarrow (\hat{\delta}_1(s_1, w), \hat{\delta}_2(s_2, w)) \in F_1 \times F_2 \\&\Leftrightarrow \hat{\delta}_1(s_1, w) \in F_1 \wedge \hat{\delta}_2(s_2, w) \in F_2 \\&\Leftrightarrow w \in L(M_1) \wedge w \in L(M_2) \\&\Leftrightarrow w \in L(M_1) \cap L(M_2).\end{aligned}$$



Frage: Funktioniert die Produktkonstruktion für den Durchschnitt auch bei NFAs?

Definition 46

Die **Umkehrung**(Spiegelung) eines Wortes $w = a_1 \cdots a_n$ ist

$$w^R := a_n \cdots a_1.$$

Die Umkehrung einer Sprache L ist

$$L^R := \{w^R; w \in L\}.$$

Satz 47

Ist L eine reguläre Sprache, dann auch L^R .

Beweis:

Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein DFA mit $L = L(M)$. Wir konstruieren einen ϵ -NFA $N = (Q \uplus \{q'_0\}, \Sigma, \delta', q'_0, \{q_0\})$ wie folgt:

- wir kehren alle Übergänge um, d.h., $\delta(q, a) = p$ gdw $q \in \delta'(p)$;
- wir fügen einen **neuen** Startzustand q'_0 hinzu, mit ϵ -Übergängen zu allen $f \in F$;
- wir machen q_0 zum (alleinigen) Endzustand von N .

Indem man die Folge der Übergänge von M bei einer beliebigen Eingabe $w \in \Sigma^*$ **rückwärts** verfolgt, ist nun leicht zu sehen, dass

$$L(N) = L^R.$$



Definition 48

Substitution (mit regulären Mengen) ist eine Abbildung, die jedem $a \in \Sigma$ eine reguläre Sprache $h(a)$ zuordnet. Diese Abbildung wird kanonisch auf Σ^* erweitert.

Ein Homomorphismus ist eine Substitution, so dass für alle $a \in \Sigma$ die Menge $h(a)$ genau ein Wort enthält, also $|h(a)| = 1$.

Satz 49

Reguläre Sprachen sind unter (regulärer) Substitution, Homomorphismus und inversem Homomorphismus abgeschlossen.

Beweis:

Wir zeigen (nur) die Behauptung für den inversen Homomorphismus.

Sei $h : \Delta \rightarrow \Sigma^*$ ein Homomorphismus, und sei $R \subseteq \Sigma^*$ regulär.

Zu zeigen: $h^{-1}(R) \subseteq \Delta^*$ ist regulär.

Sei $A = (Q, \Sigma, \delta, q_0, F)$, $L(A) = R$.

Betrachte $A' = (Q, \Delta, \delta', q_0, F)$, mit

$$\delta'(q, a) = \hat{\delta}(q, h(a)) \quad \forall q \in Q, a \in \Delta .$$

Also gilt

$$\hat{\delta}'(q_0, w) = \hat{\delta}(q_0, h(w)) \in F \Leftrightarrow h(w) \in R \Leftrightarrow w \in h^{-1}(R)$$



Definition 50

Seien $L_1, L_2 \subseteq \Sigma^*$. Dann ist der **Rechtsquotient**

$$L_1/L_2 := \{x \in \Sigma^*; (\exists y \in L_2)[xy \in L_1]\} .$$

Satz 51

Seien $R, L \subseteq \Sigma^*$, R regulär. Dann ist R/L regulär.

Beweis:

Sei A DFA mit $L(A) = R$, $A = (Q, \Sigma, \delta, q_0, F)$.

$$\begin{aligned} F' &:= \{q \in Q; (\exists y \in L)[\hat{\delta}(q, y) \in F]\} \\ A' &:= (Q, \Sigma, \delta, q_0, F') \end{aligned}$$

Dann ist $L(A') = R/L$. □

Lemma 52

Es gibt einen Algorithmus, der für zwei (nichtdeterministische, mit ϵ -Übergängen) endliche Automaten A_1 und A_2 entscheidet, ob sie äquivalent sind, d.h. ob

$$L(A_1) = L(A_2) .$$

Beweis:

Konstruiere einen endlichen Automaten für $(L(A_1) \setminus L(A_2)) \cup (L(A_2) \setminus L(A_1))$ (symmetrische Differenz). Prüfe, ob dieser Automat ein Wort akzeptiert. □

Satz 53 (Pumping Lemma für reguläre Sprachen)

Sei $R \subseteq \Sigma^*$ regulär. Dann gibt es ein $n > 0$, so dass für jedes $z \in R$ mit $|z| \geq n$ es $u, v, w \in \Sigma^*$ gibt, so dass gilt:

- 1 $z = uvw$,
- 2 $|uv| \leq n$,
- 3 $|v| \geq 1$, und
- 4 $\forall i \geq 0 : uv^i w \in R$.

Beweis:

Sei $R = L(A)$, $A = (Q, \Sigma, \delta, q_0, F)$.

Sei $n = |Q|$. Sei nun $z \in R$ mit $|z| \geq n$.

Sei $q_0 = q^{(0)}, q^{(1)}, q^{(2)}, \dots, q^{(|z|)}$ die beim Lesen von z durchlaufene Folge von Zuständen von A . Dann muss es $0 \leq i < j \leq n \leq |z|$ geben mit $q^{(i)} = q^{(j)}$.

Seien nun u die ersten i Zeichen von z , v die nächsten $j - i$ Zeichen und w der Rest.

$$\Rightarrow z = uvw, |v| \geq 1, |uv| \leq n, uv^l w \in R \quad \forall l \geq 0.$$



Beispiel für die Anwendung des Pumping Lemmas:

Satz 54

$L = \{0^{m^2}; m \geq 0\}$ ist nicht regulär.

Beweis:

Angenommen, L sei doch regulär.

Sei n wie durch das Pumping Lemma gegeben. Wähle $m \geq n$. Dann gibt es ein r mit $1 \leq r \leq n$, so dass gilt:

$$0^{m^2+ir} \in L \text{ für alle } i \in \mathbb{N}_0 .$$

Aber:

$$m^2 < m^2 + r \leq m^2 + m < m^2 + 2m + 1 = (m + 1)^2 !$$



Denkaufgabe:

$\{a^i b^i; i \geq 0\}$ ist nicht regulär.

Definition 55

Sei $L \subseteq \Sigma^*$ eine Sprache. Definiere die Relation $\equiv_L \subseteq \Sigma^* \times \Sigma^*$ durch

$$x \equiv_L y \Leftrightarrow (\forall z \in \Sigma^*) [xz \in L \Leftrightarrow yz \in L]$$

Lemma 56

\equiv_L ist eine rechtsinvariante Äquivalenzrelation.

Dabei bedeutet **rechtsinvariant**:

$$x \equiv_L y \Rightarrow xu \equiv_L yu \text{ für alle } u .$$

Beweis:

Klar! □

Satz 57 (Myhill-Nerode)

Sei $L \subseteq \Sigma^*$. Dann sind äquivalent:

- 1 L ist regulär
- 2 \equiv_L hat endlichen *Index* (= Anzahl der Äquivalenzklassen)
- 3 L ist die Vereinigung einiger der endlich vielen Äquivalenzklassen von \equiv_L .

Beweis:

(1) \Rightarrow (2):

Sei $L = L(A)$ für einen DFA $A = (Q, \Sigma, \delta, q_0, F)$.

Dann gilt

$$\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y) \Rightarrow x \equiv_L y .$$

Also gibt es höchstens so viele Äquivalenzklassen, wie der Automat A Zustände hat.

Beweis:

(2) \Rightarrow (3):

Sei $[x]$ die Äquivalenzklasse von x , $y \in [x]$ und $x \in L$.

Dann gilt nach der Definition von \equiv_L :

$$y \in L$$

Beweis:

(3) \Rightarrow (1):

Definiere $A' = (Q', \Sigma, \delta', q'_0, F')$ mit

$$\begin{aligned}Q' &:= \{[x]; x \in \Sigma^*\} && (Q' \text{ endlich!}) \\q'_0 &:= [\epsilon] \\ \delta'([x], a) &:= [xa] \quad \forall x \in \Sigma^*, a \in \Sigma && (\text{konsistent!}) \\F' &:= \{[x]; x \in L\}\end{aligned}$$

Dann gilt:

$$L(A') = L$$



3.9 Konstruktion minimaler endlicher Automaten

Satz 58

Der nach dem Satz von Myhill-Nerode konstruierte deterministische endliche Automat hat unter allen DFA's für L eine minimale Anzahl von Zuständen.

Beweis:

Sei $A = (Q, \Sigma, \delta, q_0, F)$ mit $L(A) = L$. Dann liefert

$$x \equiv_A y \Leftrightarrow \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$$

eine Äquivalenzrelation, die \equiv_L verfeinert.

Also gilt: $|Q| = \text{index}(\equiv_A) \geq \text{index}(\equiv_L) = \text{Anzahl der Zustände des Myhill-Nerode-Automaten.}$



Algorithmus zur Konstruktion eines minimalen FA

Eingabe: $A(Q, \Sigma, \delta, q_0, F)$ DFA ($L = L(A)$)

Ausgabe: Äquivalenzrelation auf Q .

- 0 Entferne aus Q alle überflüssigen, d.h. alle von q_0 aus nicht erreichbaren Zustände. Wir nehmen nun an, dass Q keine überflüssigen Zustände mehr enthält.
- 1 Markiere alle Paare $\{q_i, q_j\} \in Q^2$ mit

$$q_i \in F \text{ und } q_j \notin F \text{ bzw. } q_i \notin F \text{ und } q_j \in F .$$

- ② **for** alle unmarkierten Paare $\{q_i, q_j\} \in Q^2, q_i \neq q_j$ **do**
 if $(\exists a \in \Sigma)[\{\delta(q_i, a), \delta(q_j, a)\}$ ist markiert] **then**
 markiere $\{q_i, q_j\}$;
 for alle $\{q, q'\}$ in $\{q_i, q_j\}$'s Liste **do**
 markiere $\{q, q'\}$ und lösche aus Liste;
 ebenso rekursiv alle Paare in der Liste von $\{q, q'\}$ usw.
 od
 else
 for alle $a \in \Sigma$ **do**
 if $\delta(q_i, a) \neq \delta(q_j, a)$ **then**
 trage $\{q_i, q_j\}$ in die Liste von $\{\delta(q_i, a), \delta(q_j, a)\}$ ein
 fi
 od
 fi
od
- ③ Ausgabe: q äquivalent zu $q' \Leftrightarrow \{q, q'\}$ *nicht* markiert.

Satz 59

Obiger Algorithmus liefert einen minimalen DFA für $L(A)$.

Beweis:

Sei $A' = (Q', \Sigma', \delta', q'_0, F')$ der konstruierte Äquivalenzklassenautomat.

Offensichtlich ist $L(A) = L(A')$.

Es gilt: $\{q, q'\}$ wird markiert gdw

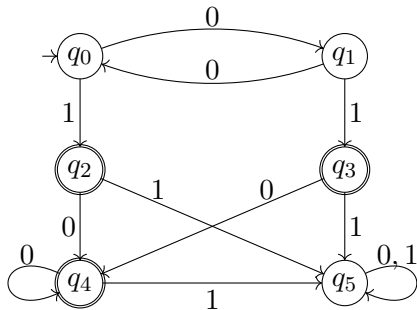
$$(\exists w \in \Sigma^*)[\hat{\delta}(q, w) \in F \wedge \hat{\delta}(q', w) \notin F \text{ oder umgekehrt}],$$

wie man durch einfache Induktion über $|w|$ sieht.

Also: Die Anzahl der Zustände von A' (nämlich $|Q'|$) ist gleich dem Index von \equiv_L . \square

Beispiel 60

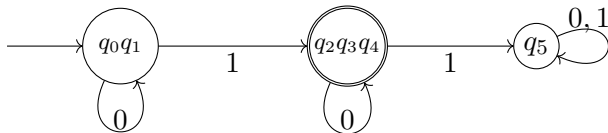
Automat A:



	q_0	q_1	q_2	q_3	q_4	q_5
q_0	/	/	/	/	/	/
q_1		/	/	/	/	/
q_2	×	×	/	/	/	/
q_3	×	×		/	/	/
q_4	×	×			/	/
q_5	×	×	×	×	×	/

Automat A' :

$$L(A') = 0^*10^*$$



Satz 61

Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein DFA. Der Zeitaufwand des obigen Minimalisierungsalgorithmus ist $O(|Q|^2|\Sigma|)$.

Beweis:

Für jedes $a \in \Sigma$ muss jede Position in der Tabelle nur konstant oft besucht werden. \square

3.10 Entscheidbarkeit

Beispiel 62

Wie wir bereits wissen, ist das Wortproblem für reguläre Grammatiken entscheidbar. Wenn L durch einen deterministischen endlichen Automaten gegeben ist, ist dies (bei festem Alphabet Σ) sogar in linearer Laufzeit möglich. Allerdings gilt, dass die Überführung eines nichtdeterministischen endlichen Automaten in einen deterministischen endlichen Automaten exponentielle Komplexität haben kann.

Die folgenden Probleme sind für Chomsky-3-Sprachen (also die Klasse der regulären Sprachen) entscheidbar:

Wortproblem: Ist ein Wort w in $L(G)$ (bzw. $L(A)$)?

Das Wortproblem ist für alle Grammatiken mit einem Chomsky-Typ größer 0 entscheidbar. Allerdings kann die Laufzeit exponentiell mit der Wortlänge n wachsen.

Für Chomsky-2- und Chomsky-3-Sprachen (d.h. -Grammatiken) gibt es wesentlich effizientere Algorithmen.

Leerheitsproblem: Ist $L(G) = \emptyset$?

Das Leerheitsproblem ist für Grammatiken vom Chomsky-Typ 2 und 3 entscheidbar.

Für andere Typen lassen sich Grammatiken konstruieren, für die nicht mehr entscheidbar ist, ob die Sprache leer ist.

Endlichkeitsproblem: Ist $|L(G)| < \infty$?

Das Endlichkeitsproblem ist für alle regulären Grammatiken lösbar.

Lemma 63

Sei n eine geeignete Pumping-Lemma-Zahl, die zur regulären Sprache L gehört. Dann gilt:

$$|L| = \infty \text{ gdw } (\exists z \in L)[n \leq |z| < 2n] .$$

Beweis:

Wir zeigen zunächst \Leftarrow :

Aus dem Pumping-Lemma folgt: $z = uvw$ für $|z| \geq n$ und $uv^i w \in L$ für alle $i \in \mathbb{N}_0$.

Damit erzeugt man unendlich viele Wörter.

Nun wird \Rightarrow gezeigt:

Dass es ein Wort z mit $|z| \geq n$ gibt, ist klar (es gibt ja unendlich viele Wörter). Mit Hilfe des Pumping-Lemmas lässt sich ein solches Wort auf eine Länge $< 2n$ reduzieren. □

Damit kann das Endlichkeitsproblem auf das Wortproblem zurückgeführt werden.

Schnittproblem: Ist $L(G_1) \cap L(G_2) = \emptyset$?

Das Schnittproblem ist für die Klasse der regulären Grammatiken entscheidbar, nicht aber für die Klasse der Chomsky-2-Grammatiken.

Äquivalenzproblem: Ist $L(G_1) = L(G_2)$?

Das Äquivalenzproblem lässt sich auch wie folgt formulieren:

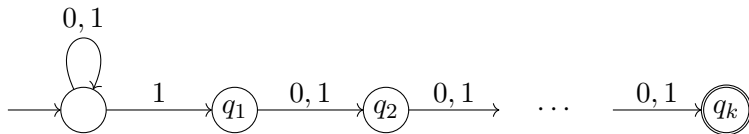
$$L_1 = L_2 \quad \Leftrightarrow \quad (L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1}) = \emptyset$$

Wichtig für eine effiziente Lösung der Probleme ist, wie die Sprache gegeben ist.
Hierzu ein Beispiel:

Beispiel 64

$L = \{w \in \{0,1\}^*; \text{das } k\text{-letzte Bit von } w \text{ ist gleich } 1\}$

Ein NFA für diese Sprache ist gegeben durch:



Insgesamt hat der NFA $k + 1$ Zustände. Man kann nun diesen NFA in einen deterministischen Automaten umwandeln und stellt fest, dass der entsprechende DFA $\Omega(2^k)$ Zustände hat.

Da die Komplexität eines Algorithmus von der Größe der Eingabe abhängt, ist dieser Unterschied in der Eingabegröße natürlich wesentlich, denn es gilt:

kurze Eingabe wie beim NFA \Rightarrow wenig Zeit für einen effizienten Algorithmus,

lange Eingabe wie beim DFA \Rightarrow mehr Zeit für einen effizienten Algorithmus.

Es gilt allerdings, dass sich für das Wortproblem (uniform oder nicht-uniform) kein großer Komplexitätsunterschied in Abhängigkeit davon ergibt, ob die Sprache durch einen NFA oder DFA dargestellt ist.

4. Kontextfreie Grammatiken und Sprachen

4.1 Grundlagen und ein Beispiel

Sei

$$L_{=} := \{w \in \{0, 1\}^*; w \text{ enthält gleich viele 0en und 1en}\}.$$

Sei $\#_a(w)$ die Anzahl der Zeichen a in der Zeichenreihe w , d.h.

$$L_{=} = \{w \in \{0, 1\}^*; \#_0(w) = \#_1(w)\}.$$

$L_{=}$ ist sicherlich nicht regulär (vgl. Pumping-Lemma).

Satz 65

Die (kontextfreie) Grammatik G

$$S \rightarrow \epsilon \mid T$$

$$T \rightarrow TT \mid 0T1 \mid 1T0 \mid 01 \mid 10$$

erzeugt $L_{=}$.

Beweis:

Sei $w \in L_{=}$. Betrachte für jedes Präfix x von w die Zahl

$$\#_1(x) - \#_0(x).$$

Falls $w = w'w''$ für nichtleere $w', w'' \in L_{=}$, wende man Induktion über $|w|$ an, falls nicht, ist w von der Form $0w'1$ oder $1w'0$, und Induktion liefert wiederum die Behauptung. □

Definition (Wiederholung, siehe Def. 23)

- Eine kontextfreie Grammatik G heißt *eindeutig*, wenn es für jedes $w \in L(G)$ genau einen Ableitungsbaum gibt.
- Eine kontextfreie Sprache L heißt *eindeutig*, falls es eine eindeutige kontextfreie Grammatik G mit $L = L(G)$ gibt. Ansonsten heißt L *inhärent mehrdeutig*.

Die oben angegebene Grammatik für $L_{=}$ ist nicht eindeutig.

4.2 Die Chomsky-Normalform

Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik.

Definition 66

Eine kontextfreie Grammatik G ist in **Chomsky-Normalform**, falls alle Produktionen eine der Formen

$$A \rightarrow a$$

$$A \in V, a \in \Sigma,$$

$$A \rightarrow BC$$

$$A, B, C \in V, \text{ oder}$$

$$S \rightarrow \epsilon$$

haben.

Algorithmus zur Konstruktion einer (äquivalenten) Grammatik in Chomsky-Normalform

Eingabe: Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$

- 1 Wir fügen für jedes $a \in \Sigma$ zu V ein neues Nichtterminal Y_a hinzu, ersetzen in allen Produktionen a durch Y_a und fügen $Y_a \rightarrow a$ als neue Produktion zu P hinzu.

/* linearer Zeitaufwand, Größe vervierfacht sich höchstens */

- 2 Wir ersetzen jede Produktion der Form

$$A \rightarrow B_1 B_2 \cdots B_r \quad (r \geq 3)$$

durch

$$A \rightarrow B_1 C_2, C_2 \rightarrow B_2 C_3, \dots, C_{r-1} \rightarrow B_{r-1} B_r,$$

wobei C_2, \dots, C_{r-1} neue Nichtterminale sind.

/* linearer Zeitaufwand, Größe vervierfacht sich höchstens */

- 3 Für alle $C, D \in V$, $C \neq D$, mit

$$C \rightarrow^+ D,$$

füge für jede Produktion der Form

$$A \rightarrow BC \in P \text{ bzw. } A \rightarrow CB \in P$$

die Produktion

$$A \rightarrow BD \text{ bzw. } A \rightarrow DB$$

zu P hinzu.

/* quadratischer Aufwand **pro** A */

- 4 Für alle $\alpha \in V^2 \cup \Sigma$, für die $S \rightarrow^* \alpha$, füge $S \rightarrow \alpha$ zu P hinzu.
- 5 Streiche alle Produktionen der Form $A \rightarrow B$ aus P .

Zusammenfassend können wir festhalten:

Satz 67

Aus einer kontextfreien Grammatik $G = (V, \Sigma, P, S)$ der Größe $s(G)$ kann in Zeit $O(|V|^2 \cdot s(G))$ eine äquivalente kontextfreie Grammatik in Chomsky-Normalform der Größe $O(|V|^2 \cdot s(G))$ erzeugt werden.

4.3 Der Cocke-Younger-Kasami-Algorithmus

Der CYK-Algorithmus (oft auch Cocke-Kasami-Younger, CKY) entscheidet das **Wortproblem** für kontextfreie Sprachen, falls die Sprache in Form einer Grammatik in Chomsky-Normalform gegeben ist.

Eingabe: Grammatik $G = (V, \Sigma, P, S)$ in Chomsky-Normalform, $w = w_1 \dots w_n \in \Sigma^*$ mit der Länge n . O.B.d.A. $n > 0$.

Definition

$$V_{ij} := \{A \in V; A \rightarrow^* w_i \dots w_j\}.$$

Es ist klar, dass $w \in L(G) \Leftrightarrow S \in V_{1n}$.

Der CYK-Algorithmus berechnet alle V_{ij} induktiv nach wachsendem $j - i$. Den Anfang machen die

$$V_{ii} := \{A \in V; A \rightarrow w_i \in P\},$$

der rekursive Aufbau erfolgt nach der Regel

$$V_{ij} = \bigcup_{i \leq k < j} \{A \in V; (A \rightarrow BC) \in P \wedge B \in V_{ik} \wedge C \in V_{k+1,j}\} \quad \text{für } i < j.$$

Die Korrektheit dieses Aufbaus ist klar, wenn die Grammatik in Chomsky-Normalform vorliegt.

Zur Komplexität des CYK-Algorithmus

Es werden $\frac{n^2+n}{2}$ Mengen V_{ij} berechnet. Für jede dieser Mengen werden $|P|$ Produktionen und höchstens n Werte für k betrachtet.

Der Test der Bedingung $(A \rightarrow BC) \in P \wedge B \in V_{ik} \wedge C \in V_{k+1,j}$ erfordert bei geeigneter Repräsentation der Mengen V_{ij} konstanten Aufwand. Der Gesamtaufwand ist also $O(|P|n^3)$.

Mit der gleichen Methode und dem gleichen Rechenaufwand kann man zu dem getesteten Wort, falls es in der Sprache ist, auch gleich einen Ableitungsbaum konstruieren, indem man sich bei der Konstruktion der V_{ij} nicht nur merkt, welche Nichtterminale sie enthalten, sondern auch gleich, warum sie sie enthalten, d.h. aufgrund welcher Produktionen sie in die Menge aufgenommen wurden.

4.4 Das Pumping-Lemma und Ogdens Lemma für kontextfreie Sprachen

Zur Erinnerung: Das Pumping-Lemma für reguläre Sprachen: Für jede reguläre Sprache L gibt es eine Konstante $n \in \mathbb{N}$, so dass sich jedes Wort $z \in L$ mit $|z| \geq n$ zerlegen lässt in $z = uvw$ mit $|uv| \leq n$, $|v| \geq 1$ und $uv^*w \subseteq L$.

Zum Beweis haben wir $n = |Q|$ gewählt, wobei Q die Zustandsmenge eines L erkennenden DFA war. Das Argument war dann, dass beim Erkennen von z (mindestens) ein Zustand zweimal besucht werden muss und damit der dazwischen liegende Weg im Automaten beliebig oft wiederholt werden kann.

Völlig gleichwertig kann man argumentieren, dass bei der Ableitung von z mittels einer rechtslinearen Grammatik ein Nichtterminalsymbol (mindestens) zweimal auftreten muss und die dazwischen liegende Teibleitung beliebig oft wiederholt werden kann.

Genau dieses Argument kann in ähnlicher Form auch auf kontextfreie Grammatiken (in Chomsky-Normalform) angewendet werden:

Satz 68 (Pumping-Lemma)

Für jede kontextfreie Sprache L gibt es eine Konstante $n \in \mathbb{N}$, so dass sich jedes Wort $z \in L$ mit $|z| \geq n$ zerlegen lässt in

$$z = uvwxy,$$

mit

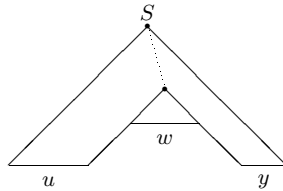
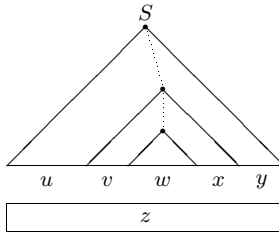
- ① $|vx| \geq 1$,
- ② $|vwx| \leq n$, *und*
- ③ $\forall i \in \mathbb{N}_0 : uv^iwx^iy \in L$.

Beweis:

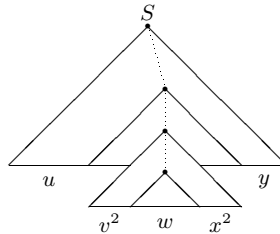
Sei $G = (V, \Sigma, P, S)$ eine Grammatik in Chomsky-Normalform mit $L(G) = L$. Wähle $n = 2^{|V|}$. Sei $z \in L(G)$ mit $|z| \geq n$. Wir zählen die Länge eines Pfades als die Anzahl seiner Knoten. Dann hat der Ableitungsbaum für z (ohne die letzte Stufe für die Terminale) mindestens die Tiefe $|V| + 1$, da er wegen der Chomsky-Normalform den Verzweigungsgrad 2 hat.

Auf einem Pfadabschnitt der Länge $\geq |V| + 1$ kommt nun mindestens ein Nichtterminal wiederholt vor. Die zwischen diesen beiden Vorkommen liegende Teibleitung kann nun beliebig oft wiederholt werden.

Beweis:



Dieser Ableitungsbaum zeigt
 $uwy \in L$



Dieser Ableitungsbaum zeigt
 $uv^2wx^2y \in L$

Beweis:

Sei $G = (V, \Sigma, P, S)$ eine Grammatik in Chomsky-Normalform mit $L(G) = L$. Wähle $n = 2^{|V|}$. Sei $z \in L(G)$ mit $|z| \geq n$. Wir zählen die Länge eines Pfades als die Anzahl seiner Knoten. Dann hat der Ableitungsbaum für z (ohne die letzte Stufe für die Terminale) mindestens die Tiefe $|V| + 1$, da er wegen der Chomsky-Normalform den Verzweigungsgrad 2 hat.

Auf einem Pfadabschnitt der Länge $\geq |V| + 1$ kommt nun mindestens ein Nichtterminal wiederholt vor. Die zwischen diesen beiden Vorkommen liegende Teibleitung kann nun beliebig oft wiederholt werden.

Um $|vwx| \leq n$ zu erreichen, muss man ein am weitesten unten liegendes Doppelvorkommen eines Nichtterminals wählen. □

Beispiel 69

Wir wollen sehen, dass die Sprache

$$\{a^i b^i c^i; i \in \mathbb{N}_0\}$$

nicht kontextfrei ist.

Wäre sie kontextfrei, so könnten wir das Wort $a^n b^n c^n$ (n die Konstante aus dem Pumping-Lemma) aufpumpen, ohne aus der Sprache herauszufallen. Wir sehen aber leicht, dass das Teilwort v nur aus a 's bestehen kann und bei jeder möglichen Verteilung des Teilworts x Pumpen entweder die Anzahl der a 's, b 's und c 's unterschiedlich ändert oder, wenn

$$(\#_a(vx) =) \#_b(vx) = \#_c(vx) > 0 ,$$

dass b 's und c 's in der falschen Reihenfolge auftreten.

Zur Vereinfachung von Beweisen wie in dem gerade gesehenen Beispiel führen wir die folgende Verschärfung des Pumping-Lemmas ein:

Satz 70 (Ogdens Lemma)

Für jede kontextfreie Sprache L gibt es eine Konstante $n \in \mathbb{N}$, so dass für jedes Wort $z \in L$ mit $|z| \geq n$ die folgende Aussage gilt:

Werden in z mindestens n (beliebige) Buchstaben markiert, so lässt sich z zerlegen in

$$z = uvwxy,$$

so dass

- 1 in vx mindestens ein Buchstabe und
- 2 in vw höchstens n Buchstaben markiert sind und
- 3 $(\forall i \in \mathbb{N}_0)[uv^iwx^iy \in L]$.

Bemerkung: Das Pumping-Lemma ist eine triviale Folgerung aus Ogdens Lemma (markiere alle Buchstaben in z).

Beweis:

Sei $G = (V, \Sigma, P, S)$ eine Grammatik in Chomsky-Normalform mit $L(G) = L$. Wähle $n = 2^{|V|}$. Sei $z \in L$ und seien in z mindestens n Buchstaben markiert. Wir messen wiederum die Länge eines Pfades als die Anzahl der in ihm enthaltenen Knoten. In einem Ableitungsbaum für z markieren wir alle (inneren) Knoten, deren linker *und* rechter Teilbaum *jeweils* mindestens ein markiertes Blatt enthalten. Es ist nun offensichtlich, dass es einen Pfad von der Wurzel zu einem Blatt gibt, auf dem mindestens $|V| + 1$ markierte innere Knoten liegen.

Beweis:

...

Wir betrachten die letzten $|V| + 1$ markierten inneren Knoten eines Pfades mit maximaler Anzahl markierter Knoten; nach dem Schubfachprinzip sind zwei mit demselben Nichtterminal, z.B. A , markiert. Wir nennen diese Knoten v_1 und v_2 . Seien die Blätter des Teilbaumes mit der Wurzel v_2 insgesamt mit w und die Blätter des Teilbaumes mit der Wurzel v_1 insgesamt mit vwx beschriftet. Es ist dann klar, dass die folgende Ableitung möglich ist:

$$S \rightarrow^* uAy \rightarrow^* uvAxy \rightarrow^* vwvxy.$$

Es ist auch klar, dass der Mittelteil dieser Ableitung weggelassen oder beliebig oft wiederholt werden kann.

Beweis:

...

Es bleibt noch zu sehen, dass vx mindestens einen und vwx höchstens n markierte Buchstaben enthält. Ersteres ist klar, da auch der Unterbaum von v_1 , der v_2 nicht enthält, ein markiertes Blatt haben muss.

Letzteres ist klar, da der gewählte Pfad eine maximale Anzahl von markierten inneren Knoten hatte und unterhalb von v_1 nur noch höchstens $|V|$ markierte Knoten auf diesem Pfad sein können. Der Teilbaum mit Wurzel v_1 kann also maximal $2^{|V|+1} = n$ markierte Blätter haben. Formal kann man z.B. zeigen, dass ein Unterbaum, der auf jedem Ast maximal k markierte (innere) Knoten enthält, höchstens 2^k markierte Blätter enthält. □

Beispiel 71

$$L = \{a^i b^j c^k d^l; i = 0 \text{ oder } j = k = l\}.$$

Hier funktioniert das normale Pumping-Lemma nicht, da für z mit $|z| \geq n$ entweder z mit a beginnt und dann z.B. $v \in \{a\}^+$ sein kann oder aber z nicht mit a beginnt und dann eine zulässige Zerlegung $z = uvwxy$ sehr einfach gewählt werden kann.

Sei n die Konstante aus Ogden's Lemma. Betrachte das Wort $ab^n c^n d^n$ und markiere darin $bc^n d$. Nun gibt es eine Zerlegung $ab^n c^n d^n = uvwxy$, so dass vx mindestens ein markiertes Symbol enthält und $uv^2wx^2y \in L$.

Es ist jedoch leicht zu sehen, dass dies einen Widerspruch liefert, da vx höchstens zwei verschiedene der Symbole b, c, d enthalten kann, damit beim Pumpen nicht die Reihenfolge durcheinander kommt.

Bemerkung:

Wie wir gerade gesehen haben, gilt die Umkehrung des Pumping-Lemmas nicht allgemein (d.h., aus dem Abschluss einer Sprache unter der Pumpoperation des Pumping-Lemmas folgt i.A. nicht, dass die Sprache kontext-frei ist).

Es gibt jedoch stärkere Versionen des Pumping-Lemmas, für die auch die Umkehrung gilt. Siehe dazu etwa



David S. Wise:

A strong pumping lemma for context-free languages.
Theoretical Computer Science **3**, pp. 359–369, 1976



Richard Johnsonbaugh, David P. Miller:

Converses of pumping lemmas.
ACM SIGCSE Bull. **22**(1), pp. 27–30, 1990

4.5 Algorithmen für kontextfreie Sprachen/Grammatiken

Satz 72

Sei $G = (V, \Sigma, P, S)$ kontextfrei. Dann kann die Menge V' der Variablen $A \in V$, für die gilt:

$$(\exists w \in \Sigma^*)[A \rightarrow^* w]$$

in Zeit $O(|V| \cdot s(G))$ berechnet werden.

Beweis:

Betrachte folgenden Algorithmus:

```
 $\Delta := \{A \in V; (\exists(A \rightarrow w) \in P \text{ mit } w \in \Sigma^*)\}; V' := \emptyset;$   
while  $\Delta \neq \emptyset$  do  
     $V' := V' \cup \Delta$   
     $\Delta := \{A \in V \setminus V'; (\exists A \rightarrow \alpha) \in P \text{ mit } \alpha \in (V' \cup \Sigma)^*\}$   
od
```

Induktion über die Länge der Ableitung. □

Definition 73

$A \in V$ heißt **nutzlos**, falls es keine Ableitung

$$S \rightarrow^* w, \quad w \in \Sigma^*$$

gibt, in der A vorkommt.

Satz 74

Die Menge der nutzlosen Variablen kann in Zeit $O(|V| \cdot s(G))$ bestimmt werden.

Beweis:

Sei V'' die Menge der nicht nutzlosen Variablen.

Offensichtlich gilt: $V'' \subseteq V'$ (V' aus dem vorigen Satz).

Falls $S \notin V'$, dann sind alle Variablen nutzlos.

Ansonsten:

$\Delta := \{S\}; V'' := \emptyset;$

while $\Delta \neq \emptyset$ **do**

$V'' := V'' \cup \Delta$

$\Delta := \{B \in V' \setminus V''; (\exists A \rightarrow \alpha B \beta) \in P \text{ mit } A \in V'',$
 $\alpha, \beta \in (V' \cup \Sigma)^*\}$

od

Induktion über Länge der Ableitung: Am Ende des Algorithmus ist V'' gleich der Menge der nicht nutzlosen Variablen. □

Bemerkung: Alle nutzlosen Variablen und alle Produktionen, die nutzlose Variablen enthalten, können aus der Grammatik entfernt werden, ohne die erzeugte Sprache zu ändern.

Korollar 75

Für eine kontextfreie Grammatik G kann in Zeit $O(|V| \cdot s(G))$ entschieden werden, ob $L(G) = \emptyset$.

Beweis:

$$L(G) = \emptyset \iff S \notin V'' \text{ (bzw. } S \notin V')$$

□

Satz 76

Für eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ ohne nutzlose Variablen und in Chomsky-Normalform kann in linearer Zeit entschieden werden, ob

$$|L(G)| < \infty.$$

Beweis:

Definiere gerichteten Hilfsgraphen mit Knotenmenge V und

$$\text{Kante } A \rightarrow B \iff (A \rightarrow BC) \text{ oder } (A \rightarrow CB) \in P.$$

$L(G)$ ist endlich \iff dieser Digraph enthält keinen Zyklus.

Verwende DFS, um in linearer Zeit festzustellen, ob der Digraph Zyklen enthält. \square

Satz 77

Seien kontextfreie Grammatiken $G_1 = (V_1, \Sigma_1, P_1, S_1)$ und $G_2 = (V_2, \Sigma_2, P_2, S_2)$ gegeben. Dann können in linearer Zeit kontextfreie Grammatiken für

- 1 $L(G_1) \cup L(G_2)$,
- 2 $L(G_1)L(G_2)$,
- 3 $(L(G_1))^*$

konstruiert werden. Die Klasse der kontextfreien Sprachen ist also unter *Vereinigung*, *Konkatenation* und *Kleene'scher Hülle* abgeschlossen.

Beweis:

Ohne Beschränkung der Allgemeinheit nehmen wir an, dass $V_1 \cap V_2 = \emptyset$.

- 1 $V = V_1 \cup V_2 \cup \{S\}$; S neu
 $P = P_1 \cup P_2 \cup \{S \rightarrow S_1|S_2\}$
- 2 $V = V_1 \cup V_2 \cup \{S\}$; S neu
 $P = P_1 \cup P_2 \cup \{S \rightarrow S_1S_2\}$
- 3 $V = V_1 \cup \{S, S'\}$; S, S' neu
 $P = P_1 \cup \{S \rightarrow S'|\epsilon, S' \rightarrow S_1S'|S_1\}$

Falls $\epsilon \in L(G_1)$ oder $\epsilon \in L(G_2)$, sind noch Korrekturen vorzunehmen, die hier als Übungsaufgabe überlassen bleiben. □

Satz 78

Die Klasse der kontextfreien Sprachen ist *nicht* abgeschlossen unter Durchschnitt oder Komplement.

Beweis:

Es genügt zu zeigen (wegen **de Morgan** (1806–1871)): nicht abgeschlossen unter Durchschnitt.

$L_1 := \{a^i b^i c^j; i, j \geq 0\}$ ist kontextfrei

$L_2 := \{a^i b^j c^j; i, j \geq 0\}$ ist kontextfrei

$L_1 \cap L_2 = \{a^i b^i c^i; i \geq 0\}$ ist nicht kontextfrei



Satz 79

Die Klasse der kontextfreien Sprachen ist abgeschlossen gegenüber Substitution (mit kontextfreien Mengen).

Beweis:

Ersetze jedes Terminal a durch ein neues Nichtterminal S_a und füge zu den Produktionen P für jedes Terminal a die Produktionen einer kontextfreien Grammatik $G_a = (V_a, \Sigma, P_a, S_a)$ hinzu.

Forme die so erhaltene Grammatik in eine äquivalente Chomsky-2-Grammatik um. \square

4.6 Greibach-Normalform

Definition 80

Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. G ist in **Greibach-Normalform** (benannt nach **Sheila Greibach** (UCLA)), falls jede Produktion $\neq S \rightarrow \epsilon$ von der Form

$$A \rightarrow a\alpha \text{ mit } a \in \Sigma, \alpha \in V^*$$

ist.

Lemma 81

Sei $G = (V, \Sigma, P, S)$ kontextfrei, $(A \rightarrow \alpha_1 B \alpha_2) \in P$, und sei $B \rightarrow \beta_1 | \beta_2 | \dots | \beta_r$ die Menge der B -Produktionen (also die Menge der Produktionen mit B auf der linken Seite). **Ersetzt** man $A \rightarrow \alpha_1 B \alpha_2$ durch $A \rightarrow \alpha_1 \beta_1 \alpha_2 | \alpha_1 \beta_2 \alpha_2 | \dots | \alpha_1 \beta_r \alpha_2$, so ändert sich die von der Grammatik erzeugte Sprache nicht.

Lemma 82

Sei $G = (V, \Sigma, P, S)$ kontextfrei, sei $A \rightarrow A\alpha_1|A\alpha_2|\dots|A\alpha_r$ die Menge der linksrekursiven A -Produktionen (alle $\alpha_i \neq \epsilon$, die Produktion $A \rightarrow A$ kommt o.B.d.A. nicht vor), und seien $A \rightarrow \beta_1|\beta_2|\dots|\beta_s$ die restlichen A -Produktionen (ebenfalls alle $\beta_i \neq \epsilon$).

Ersetzen wir alle A -Produktionen durch

$$\begin{aligned} A &\rightarrow \beta_1|\dots|\beta_s|\beta_1A'|\dots|\beta_sA' \\ A' &\rightarrow \alpha_1|\dots|\alpha_r|\alpha_1A'|\dots|\alpha_rA', \end{aligned}$$

wobei A' ein neues Nichtterminal ist, so ändert sich die Sprache nicht, und die neue Grammatik enthält keine linksrekursive A -Produktion mehr.

Beweis:

Von A lassen sich vor der Transformation alle Zeichenreihen der Form

$$(\beta_1|\beta_2|\dots|\beta_s)(\alpha_1|\alpha_2|\dots|\alpha_r)^*$$

ableiten.

Dies ist auch nach der Transformation der Fall. Während vor der Transformation alle Zeichenreihen der obigen Form von **rechts** her aufgebaut werden, werden sie danach von **links** nach rechts erzeugt.

Die Umkehrung gilt ebenso. □

Satz 83

Zu jeder kontextfreien Grammatik G gibt es eine äquivalente Grammatik in Greibach-Normalform.

Beweis:

Sei o.B.d.A. $G = (V, \Sigma, P, S)$ mit $V = \{A_1, \dots, A_m\}$ in Chomsky-Normalform und enthalte keine nutzlosen Variablen.

Bemerkung: Im folgenden Algorithmus werden ggf neue Variablen hinzugefügt, die Programmvariable m ändert sich dadurch entsprechend!

Beweis:

```
for  $k = 1, \dots, m$  do  
  for  $j = 1, \dots, k - 1$  do  
    for all  $(A_k \rightarrow A_j \alpha) \in P$  do  
      ersetze die Produktion gemäß der Konstruktion  
      in Lemma 81
```

od

od

```
co die rechte Seite keiner  $A_k$ -Produktion beginnt nun  
    noch mit einer Variablen  $A_j, j < k$  oc
```

ersetze alle linksrekursiven A_k -Produktionen gemäß der
Konstruktion in Lemma 82

```
co die rechte Seite keiner  $A_k$ -Produktion beginnt nun  
    noch mit einer Variablen  $A_j, j \leq k$  oc
```

od

Beweis (Forts.):

Da nun für jede Produktion der Form

$$A_k \rightarrow A_j \alpha$$

gilt:

$$j > k$$

(dies impliziert insbesondere, dass die rechte Seite jeder A_m -Produktion mit einem Terminalzeichen beginnt), können wir durch genügend oftmalige Anwendung der Konstruktion in Lemma 81 erreichen, dass jede rechte Seite mit einem Terminalzeichen beginnt. □

Korollar 84

Sei G eine kontextfreie Grammatik. Es gibt einen Algorithmus, der eine zu G äquivalente Grammatik in Greibach-Normalform konstruiert, deren rechte Seiten jeweils höchstens zwei Variablen enthalten.

Beweis:

Klar!



4.7 Kellerautomaten

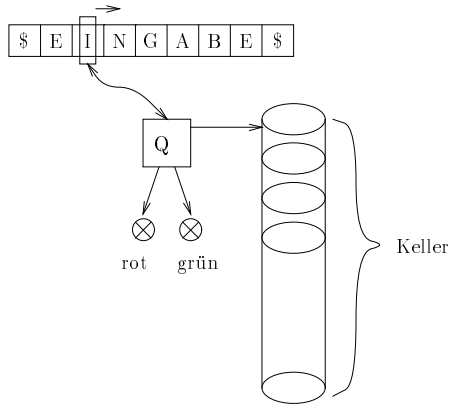
In der Literatur findet man häufig auch die Bezeichnungen **Stack-Automat** oder **Pushdown-Automat**. Kellerautomaten sind, wenn nichts anderes gesagt wird, **nichtdeterministisch**.

Definition 85

Ein NPDA = PDA (= Nichtdeterministischer Pushdown-Automat) besteht aus:

Q	endliche Zustandsmenge
Σ	endliches Eingabealphabet
Δ	endliches Stackalphabet
$q_0 \in Q$	Anfangszustand
$Z_0 \in \Delta$	Initialisierung des Stack
δ	Übergangsrelation Fkt. $Q \times (\Sigma \cup \{\epsilon\}) \times \Delta \rightarrow 2^{Q \times \Delta^*}$ wobei $ \delta(q, a, Z) + \delta(q, \epsilon, Z) < \infty \quad \forall q, a, Z$
$F \subseteq Q$	akzeptierende Zustände

Der Kellerautomat



Konfiguration:

Tupel (q, w, α) mit

$$\begin{aligned} q &\in Q \\ w &\in \Sigma^* \\ \alpha &\in \Delta^* \end{aligned}$$

Schritt:

$$(q, w_0 w', Z \alpha') \rightarrow (q', w', Z_1 \dots Z_r \alpha')$$

wenn $(q', Z_1 \dots Z_r) \in \delta(q, w_0, Z)$

bzw.:

$$(q, w, Z \alpha') \rightarrow (q', w, Z_1 \dots Z_r \alpha')$$

wenn $(q', Z_1 \dots Z_r) \in \delta(q, \epsilon, Z)$

Definition 86

- ① Ein NPDA A akzeptiert $w \in \Sigma^*$ durch leeren Stack, falls

$$(q_0, w, Z_0) \rightarrow^* (q, \epsilon, \epsilon) \text{ für ein } q \in Q .$$

- ② Ein NPDA A akzeptiert $w \in \Sigma^*$ durch akzeptierenden Zustand, falls

$$(q_0, w, Z_0) \rightarrow^* (q, \epsilon, \alpha) \text{ für ein } q \in F, \alpha \in \Delta^* .$$

- ③ Ein NPDA heißt **deterministisch (DPDA)**, falls

$$|\delta(q, a, Z)| + |\delta(q, \epsilon, Z)| \leq 1 \quad \forall (q, a, Z) \in Q \times \Sigma \times \Delta .$$

Beispiel 87

Der PDA mit

$$\delta(q_0, a, *) = \{(q_0, a*)\} \quad \text{für } a \in \{0, 1\}, * \in \{0, 1, Z_0\}$$

$$\delta(q_0, \#, *) = \{(q_1, *)\} \quad \text{für } * \in \{0, 1, Z_0\}$$

$$\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$$

$$\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$$

$$\delta(q_1, \epsilon, Z_0) = \{(q_1, \epsilon)\} \quad \text{akzeptiert mit leerem Stack}$$

die Sprache

$$L = \{w\#w^R; w \in \{0, 1\}^*\}.$$

Beispiel 87

Der PDA mit

$$\begin{aligned}\delta(q_0, a, *) &= \{(q_0, a *)\} && \text{für } a \in \{0, 1\}, * \in \{0, 1, Z_0\} \\ \delta(q_0, \#, *) &= \{(q_1, *)\} && \text{für } * \in \{0, 1, Z_0\} \\ \delta(q_1, 0, 0) &= \{(q_1, \epsilon)\} \\ \delta(q_1, 1, 1) &= \{(q_1, \epsilon)\} \\ \delta(q_1, \epsilon, Z_0) &= \{(q_a, \epsilon)\} && \text{akzeptiert mit akzeptierendem} \\ &&& \text{Zustand } (F = \{q_a\}) \\ &&& \text{(und leerem Stack)}\end{aligned}$$

die Sprache

$$L = \{w\#w^R; w \in \{0, 1\}^*\}.$$

Satz 88

Sei $A_1 = (Q, \Sigma, \Delta, q_0, Z_0, \delta, F)$ ein NPDA, der mit akzeptierendem Zustand akzeptiert. Dann kann in linearer Zeit ein NPDA $A_2 = (Q', \Sigma, \Delta', q'_0, Z'_0, \delta')$ konstruiert werden, der $L(A_1)$ mit leerem Stack akzeptiert.

Beweis:

A_2 simuliert A_1 . Sobald A_1 in einen akzeptierenden Zustand gerät, rät A_2 , ob die Eingabe zu Ende gelesen ist. Falls A_2 dies meint, wird der Keller geleert.

Um zu verhindern, dass bei der Simulation von A_1 der Keller leer wird, ohne dass A_1 akzeptiert, führen wir ein neues Kellersymbol Z'_0 ein:

$$\begin{aligned}Q' &= Q \cup \{\bar{q}, q'_0\} \\ \Delta' &= \Delta \cup \{Z'_0\}\end{aligned}$$

und wir **erweitern** δ zu δ' gemäß

$$\delta'(q'_0, \epsilon, Z'_0) = \{(q_0, Z_0 Z'_0)\}$$

$$\delta'(q, \epsilon, Z) \cup= \{(\bar{q}, \epsilon)\} \quad \text{für } q \in F, Z \in \Delta'$$

$$\delta'(\bar{q}, \epsilon, Z) = \{(\bar{q}, \epsilon)\} \quad \text{für } Z \in \Delta'$$



Bemerkung:

Akzeptieren mit leerem Keller bedeutet, dass der NPDA akzeptiert, falls der Keller leer ist **und** die Eingabe gelesen ist.

Bemerkung:

Akzeptieren mit leerem Keller bedeutet, dass der NPDA akzeptiert, falls der Keller leer ist **und** die Eingabe gelesen ist, bzw.

dass, falls der Keller leer ist, der NPDA **die bisher gelesene Eingabe** akzeptiert.

Satz 89

Sei $A_1 = (Q, \Sigma, \Delta, q_0, Z_0, \delta)$ ein NPDA, der mit leerem Keller akzeptiert. Dann kann in linearer Zeit ein NPDA $A_2 = (Q', \Sigma, \Delta', q'_0, Z'_0, \delta', F)$ konstruiert werden, welcher $L(A_1)$ mit akzeptierendem Zustand akzeptiert.

Beweis:

A_2 simuliert A_1 . Am Anfang steht ein neues Kellersymbol auf dem Stack. Sobald bei der Simulation von A_1 dieses auf dem Stack vorgefunden wird, weiß man, dass A_1 seinen Stack leergeräumt hat und folglich akzeptiert. Folglich geht A_2 in einen akzeptierenden Zustand und hält:

$$Q' = Q \cup \{q'_0, q_f\}$$

$$\Delta' = \Delta \cup \{Z'_0\}$$

$$F = \{q_f\}$$

$$\delta'(q'_0, \epsilon, Z'_0) = \{(q_0, Z_0 Z'_0)\}$$

$$\delta'(q, a, Z) = \delta(q, a, Z) \text{ für } q \in Q, a \in \Sigma \cup \{\epsilon\}, Z \in \Delta$$

$$\delta'(q, \epsilon, Z'_0) = \{(q_f, Z'_0)\} \text{ für } q \in Q$$



4.8 Kellerautomaten und kontextfreie Sprachen

Satz 90

Sei G eine CFG in Greibach-Normalform. Dann kann in linearer Zeit ein NPDA N konstruiert werden (welcher mit leerem Stack akzeptiert), so dass

$$L(N) = L(G) .$$

Beweis:

Sei o.B.d.A. $\epsilon \notin L(G)$.

Der Automat startet mit S auf dem Stack. Er sieht sich in jedem Schritt das oberste Stacksymbol A an und überprüft, ob es in G eine Produktion gibt, deren linke Seite A ist und deren rechte Seite mit dem Terminal beginnt, welches unter dem Lesekopf steht.

Sei also $G = (V, T, P, S)$.

Konstruiere NPDA $N = (Q, \Sigma, \Delta, q_0, Z_0, \delta)$ mit

$$Q := \{q_0\}$$

$$\Delta := V$$

$$\Sigma := T$$

$$Z_0 := S$$

$$\delta(q_0, a, A) \ni (q_0, \alpha) \quad \text{für } (A \rightarrow a\alpha) \in P.$$

Beweis (Forts.):

Zu zeigen ist nun: $L(N) = L(G)$.

Hilfsbehauptung:

$$S \rightarrow_G^* w_1 \dots w_i A_1 \dots A_m \text{ mit } w_j \in T, A_j \in V \text{ per Linksableitung} \\ \Leftrightarrow (q_0, w_1 \dots w_i, Z_0) \rightarrow_N^* (q_0, \epsilon, A_1 \dots A_m)$$

Der Beweis erfolgt durch Induktion über die Anzahl der Schritte in der Linksableitung.

Beweis (Forts.):

Induktionsanfang ($i = 0$):

$$S \rightarrow_G^* S \quad \Leftrightarrow \quad (q_0, \epsilon, Z_0) \rightarrow_N^* (q_0, \epsilon, Z_0)$$

Beweis (Forts.):

Induktionsschritt $((i - 1) \mapsto i)$:

$$S \rightarrow_G^* w_1 \dots w_i A_1 \dots A_m$$

$$\Leftrightarrow S \rightarrow_G^* w_1 \dots w_{i-1} A' A_v \dots A_m \quad v \in \{1, \dots, m + 1\}$$

$$\rightarrow_G w_1 \dots w_i A_1 \dots A_m$$

$$\text{(also } (A' \rightarrow w_i A_1 \dots A_{v-1}) \in P)$$

gemäß Induktionsvoraussetzung

$$\Leftrightarrow (q_0, w_1 \dots w_{i-1}, Z_0) \rightarrow_N^* (q_0, \epsilon, A' A_v \dots A_m)$$

$$\Leftrightarrow (q_0, w_1 \dots w_{i-1} w_i, Z_0) \rightarrow_N^* (q_0, w_i, A' A_v \dots A_m)$$

$$\rightarrow_N (q_0, \epsilon, A_1 \dots A_m)$$

$$\text{da } (A' \rightarrow w_i A_1 \dots A_{v-1}) \in P)$$

$$\Leftrightarrow (q_0, w_1 \dots w_i, Z_0) \rightarrow_N^* (q_0, \epsilon, A_1 \dots A_m)$$

Beweis (Forts.):

Aus der Hilfsbehauptung folgt

$$L(N) = L(G) .$$



Satz 91

Sei $N = (Q, \Sigma, \Delta, q_0, Z_0, \delta)$ ein NPDA, der mit leerem Keller akzeptiert. Dann ist $L(N)$ kontextfrei.

Beweis:

Wir definieren:

$$G = (V, T, P, S)$$

$$T := \Sigma$$

$$V := Q \times \Delta \times Q \cup \{S\} \quad \text{wobei wir die Tupel mit } [, ,] \text{ notieren}$$

$$P \ni S \rightarrow [q_0, Z_0, q] \text{ f\u00fcr } q \in Q$$

$$P \ni [q, Z, q_m] \rightarrow a[p, Z_1, q_1][q_1, Z_2, q_2] \cdots [q_{m-1}, Z_m, q_m]$$

$$\text{f\u00fcr } \delta(q, a, Z) \ni (p, Z_1 \cdots Z_m), \forall q_1, \dots, q_m \in Q,$$

$$\text{mit } a \in \Sigma \cup \{\epsilon\}.$$

Idee: Aus $[p, X, q]$ sollen sich alle die W\u00f6rter ableiten lassen, die der NPDA N lesen kann, wenn er im Zustand p mit (lediglich) X auf dem Stack startet und im Zustand q mit leerem Stack endet.

Beweis (Forts.):

Hilfsbehauptung:

$$[p, X, q] \rightarrow_G^* w \Leftrightarrow (p, w, X) \rightarrow_N^* (q, \epsilon, \epsilon).$$

„ \Rightarrow “: Induktion über die Länge l der Ableitung.

Induktionsanfang ($l = 1$):

$$\begin{aligned} & [p, X, q] \rightarrow_G w \\ \Rightarrow & \delta(p, w, X) \ni (q, \epsilon) \\ \Rightarrow & (p, w, X) \rightarrow_N (q, \epsilon, \epsilon) \end{aligned}$$

Beweis (Forts.):

Induktionsschritt $((l - 1) \mapsto l)$:

Gelte

$$\begin{aligned} [p, X, q_{m+1}] &\rightarrow_G a[q_1, X_1, q_2][q_2, X_2, q_3] \cdots [q_m, X_m, q_{m+1}] \\ &\rightarrow_G^* aw^{(1)} \cdots w^{(m)} = w \end{aligned}$$

mit $(q_1, X_1 \cdots X_m) \in \delta(p, a, X)$, $[q_i, X_i, q_{i+1}] \rightarrow_G^{l_i} w^{(i)}$ und $\sum l_i < l$.

Dann gilt gemäß Induktionsvoraussetzung

$$\begin{aligned} \Rightarrow & (q_i, w^{(i)}, X_i) \rightarrow_N^{l_i} (q_{i+1}, \epsilon, \epsilon) \quad \forall i \in \{1, \dots, m\} \\ \Rightarrow & (p, \underbrace{aw^{(1)} \cdots w^{(m)}}_{=w}, X) \rightarrow_N (q_1, w^{(1)} \cdots w^{(m)}, X_1 \cdots X_m) \\ & \rightarrow_N^{<l} (q_{m+1}, \epsilon, \epsilon) . \end{aligned}$$

Beweis (Forts.):

„ \Leftarrow “: Induktion über die Länge l einer Rechnung des NPDA's N

Induktionsanfang ($l = 1$):

$$\begin{aligned} & (p, w, X) \rightarrow_N (q, \epsilon, \epsilon) \\ \Rightarrow & (q, \epsilon) \in \delta(p, w, X) \quad (\text{also } |w| \leq 1) \\ \Rightarrow & ([p, X, q] \rightarrow w) \in P. \end{aligned}$$

Beweis (Forts.):

Induktionsschritt $((l - 1) \mapsto l)$:

Sei

$$\begin{aligned}(p, w, X) &\rightarrow_N (q_1, w', X_1 \cdots X_m) \\ &\rightarrow_N^* (q, \epsilon, \epsilon)\end{aligned}$$

eine Rechnung von N der Länge l , mit $w = ew'$ und $e \in \Sigma \cup \{\epsilon\}$.

Nach Definition gibt es

$$([p, X, q] \rightarrow e[q_1 X_1 q_2] \cdots [q_m X_m q_{m+1}]) \in P \quad \text{mit } q_{m+1} = q$$

und eine Zerlegung $w' = w^{(1)} \cdots w^{(m)}$, so dass $w^{(1)} \cdots w^{(i)}$ der von N zu dem Zeitpunkt verarbeitete Teilstring von w' ist, wenn X_{i+1} zum ersten Mal oberstes Stacksymbol (bzw., für $i = m$, der Stack leer) wird.

Beweis (Forts.):

Gemäß Induktionsvoraussetzung gilt also

$$(q_i, w^{(i)}, X_i) \xrightarrow{N}^{l_i} (q_{i+1}, \epsilon, \epsilon) \quad \text{mit } \sum l_i < l \text{ und} \\ [q_i, X_i, q_{i+1}] \xrightarrow{G}^* w^{(i)} .$$

Also folgt:

$$[p, X, q] \xrightarrow{G} e[q_1, X_1, q_2] \cdots [q_m, X_m, q_{m+1}] \quad \text{mit } q_{m+1} = q \\ \xrightarrow{G}^{<l} ew^{(1)} \cdots w^{(m)} = w$$

Aus der Hilfsbehauptung folgt der Satz. □

Satz 92

Folgende Aussagen sind äquivalent:

- L wird von einer *kontextfreien Grammatik* erzeugt.
- L wird von einem *NPDA* akzeptiert.

Beweis:

Folgt aus den vorhergehenden Sätzen. □

4.9 Deterministische Kellerautomaten

Wir haben bereits definiert:

Ein PDA heißt **deterministisch (DPDA)**, falls

$$|\delta(q, a, Z)| + |\delta(q, \epsilon, Z)| \leq 1 \quad \forall (q, a, Z) \in Q \times \Sigma \times \Delta .$$

Die von einem DPDA, der mit **leerem Keller akzeptiert**, erkannte Sprache genügt der **Fano-Bedingung**, d.h. kein Wort in der Sprache ist echtes Präfix eines anderen Wortes in der Sprache.

Festlegung:

Da wir an einem weniger eingeschränkten Maschinenmodell interessiert sind, legen wir fest, dass ein DPDA stets mit **akzeptierenden Zuständen** akzeptiert.

Definition 93

Ein DPDA ist in **Normalform**, falls gilt:

- 1 $(q', \alpha) = \delta(q, e, X)$ für $e \in \Sigma \cup \{\epsilon\}$, $q, q' \in Q$, $X \in \Delta$
 $\Rightarrow \alpha \in \{\epsilon, X, YX\}$ für $Y \in \Delta$.
- 2 Der Automat liest jede Eingabe vollständig.

Satz 94

Zu jedem DPDA $A = (Q, \Sigma, \Delta, q_0, Z_0, \delta, F)$ kann ein äquivalenter DPDA in Normalform konstruiert werden.

Beweis:

Erste Schritte der Konstruktion:

- ① Werden von A in einem Schritt mehr als zwei Symbole auf dem Stack abgelegt, wird dies von A' durch eine Folge von Schritten mit je 2 Stacksymbolen ersetzt.
- ② Werden zwei oder ein Stacksymbol abgelegt und dabei das oberste Stacksymbol X geändert, entfernen wir zunächst in einem eigenen Schritt das oberste Stacksymbol und pushen dann die gewünschten Symbole. (Das „Merken“ erfolgt in der Zustandsmenge Q' .)
- ③ Wir vervollständigen δ' mittels eines (nicht akzeptierenden) Fangzustandes. Es könnte hier noch sein, dass der DPDA ab einem Zeitpunkt nur mehr und unbegrenzt viele ϵ -Übergänge ausführt.

Beweis (Forts.):

Hilfsbehauptung:

Der DPDA A führt ab einer bestimmten Konfiguration (q, ϵ, β) unendlich viele direkt aufeinander folgende ϵ -Übergänge genau dann aus, wenn

$$\begin{array}{l} (q, \epsilon, \beta) \rightarrow^* (q', \epsilon, X\beta') \quad \text{und} \\ (q', \epsilon, X) \rightarrow^+ (q', \epsilon, X\alpha) \quad \text{für } q, q' \in Q \\ X \in \Delta, \alpha, \beta, \beta' \in \Delta^* \end{array}$$

„ \Leftarrow “: klar

Beweis (Forts.):

„ \Rightarrow “: Betrachte eine unendlich lange Folge von ϵ -Übergängen.

Sei $n := |Q| \cdot |\Delta| + |\beta| + 1$.

Wird die Stackhöhe n nie erreicht, so muss sich sogar eine Konfiguration des DPDA's wiederholen. Daraus folgt die Behauptung.

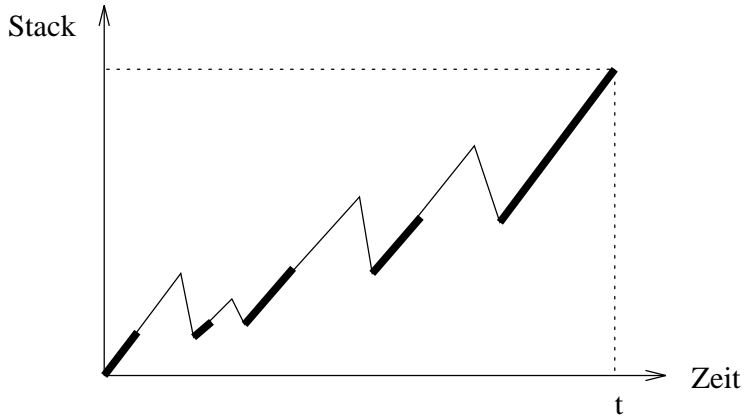
Beweis (Forts.):

Ansonsten wird jede Stackhöhe $|\beta|, \dots, n$ mindestens einmal erreicht (wegen der Normalform ist die Höhendifferenz pro Schritt $\in \{-1, 0, 1\}$).

Betrachte den Zeitpunkt t , in dem die Stackhöhe zum erstenmal n ist. Markiere für jedes $i \in \{|\beta|, \dots, n\}$ den Zeitpunkt t_i , wo zum letzten Mal (vor Zeitpunkt t) die Stackhöhe $= i$ ist. Zu diesen Zeitpunkten t_i betrachte die Paare $(q, X) \in Q \times \Delta$, wobei q der Zustand des DPDA's und X das oberste Kellersymbol des DPDA's zu diesem Zeitpunkt ist.

Da es mehr als $|\Delta| \cdot |Q|$ markierte Paare gibt, taucht ein markiertes Paar (q', X) doppelt auf. Für dieses gilt dann $(q', \epsilon, X) \rightarrow^+ (q', \epsilon, X\alpha)$.

Beweis (Forts.):



Beweis (Forts.):

Das gleiche Argument gilt, falls sich die Stackhöhe um $> |Q| \cdot |\Delta|$ erhöht.

Damit lassen sich alle Paare (q', X) finden, für die gilt:

$$(q', \epsilon, X) \rightarrow^+ (q', \epsilon, X\alpha), \alpha \in \Delta^*.$$

Da der DPDA nicht endlos weiterlaufen soll, ersetzen wir $\delta(q', \epsilon, X)$ durch einen ϵ -Übergang in einen neuen Zustand q'' (der genau dann akzeptierend ist, wenn in der Schleife $(q', \epsilon, X) \rightarrow^+ (q', \epsilon, X\alpha)$ ein akzeptierender Zustand auftritt) und einen ϵ -Übergang von q'' in den nichtakzeptierenden Fangzustand. Die Details dieser Konstruktion werden nun beschrieben.

Beweis (Forts.):

Wir modifizieren den DPDA A in mehreren Schritten wie folgt:

1. A merkt sich das oberste Kellersymbol im Zustand:

$$Q' := \{q_{\text{pop}}; q \in Q\} \cup \{qX; q \in Q, X \in \Delta\}$$

Für die Übergangsrelation δ' setzen wir (für $e \in \Sigma \cup \{\epsilon\}$)

$$\delta'(qX, e, X) := \begin{cases} (pY, YX) \\ (pX, X) \\ (p_{\text{pop}}, \epsilon) \end{cases} \quad \text{falls } \delta(q, e, X) = \begin{cases} (p, YX) \\ (p, X) \\ (p, \epsilon) \end{cases}$$

Im dritten Fall kommt noch

$$\delta'(p_{\text{pop}}, \epsilon, X) := (pX, X)$$

für alle $X \in \Delta$ dazu.

Beweis (Forts.):

Weiter

$$q'_0 := q_0 Z_0$$

$$F' := \{qX; q \in F, X \in \Delta\}$$

Ein Zustand q' heißt **spontan**, falls q' von der Form p_{pop} (und damit $\delta'(q', \epsilon, X)$ für alle $X \in \Delta$ definiert) ist oder falls

$$q' = qX$$

und $\delta'(qX, \epsilon, X)$ definiert ist.

Beweis (Forts.):

Wir erweitern Q' um einen neuen Zustand f , $f \notin F'$, der als **Fangzustand** dient:

- für alle $q' = qX$, q' nicht spontan, $a \in \Sigma$, so dass $\delta'(qX, a, X)$ nicht definiert ist, setze

$$\delta'(qX, a, X) := (f, X);$$

- setze

$$\delta'(f, a, X) := (f, X)$$

für alle $a \in \Sigma$, $X \in \Delta$.

Beweis (Forts.):

Bemerkungen:

- 1 f ist nicht spontan, f ist **kein** (akzeptierender) Endzustand.
- 2 Für alle nicht-spontanen $q' \in Q'$ von der Form $q' = qX$ ist $\delta'(q', a, X)$ für alle $a \in \Sigma$ definiert.
- 3 $\delta'(f, a, X)$ ist für alle $a \in \Sigma$ und $X \in \Delta$ definiert.

Falls sich der DPDA also in einem nicht-spontanen Zustand befindet und ein weiteres Eingabezeichen zur Verfügung steht, wird dieses gelesen!

2. *Endliche Gedächtniserweiterung*: Wir erweitern den DPDA so, dass er sich eine vorgegebene **endliche** Menge von Alternativen merken kann. Ersetzen wir z.B. Q' durch $Q' \times \{0, 1\}^m$, so kann sich der Automat Information im Umfang von m Bits (also 2^m Alternativen) merken und diese bei Übergängen fortschreiben.

Der neue Anfangszustand, die Menge der (akzeptierenden) Endzustände und die neue Übergangsrelation werden entsprechend der intendierten Semantik des endlichen Speichers festgelegt.

Sei A' der DPDA vor der „Speichererweiterung“. Wir erweitern A' zu A'' , so dass A'' sich ein zusätzliches Bit im Zustand merken kann. Dieses Bit ist im Anfangszustand von A'' gleich 0. Bei einem Zustandsübergang von A'' , der einem Zustandsübergang von A' aus einem **spontanen** Zustand q' entspricht, gilt: Ist $q' \in F'$, so wird das Bit im Zustand nach dem Übergang gesetzt, ansonsten kopiert. Entspricht der Zustandsübergang von A'' einem Zustandsübergang von A' aus einem **nicht-spontanen** Zustand, so wird das Bit gelöscht.

Beweis (Forts.):

Der Fangzustand f mit gesetztem Bit (i.Z. $f^{(1)}$) wird nun (akzeptierender) Endzustand, f mit nicht gesetztem Bit (i.Z. $f^{(0)}$) bleibt Nicht-Endzustand.

3. *Entfernung unendlicher Folgen von ϵ -Übergängen:* Für alle Zustände $q' = qX$ von A' , für die gilt

$$(q', \epsilon, X) \rightarrow^+ (q', \epsilon, X\alpha),$$

setzen wir

$$\delta'(q', \epsilon, X) := (f, X).$$

In A'' setzt dieser Übergang das Speicherbit, falls die obige Schleife einen Endzustand enthält (womit A'' in $f^{(1)}$ endet), ansonsten wird das Speicherbit kopiert.

Beweis (Forts.):

Bemerkung: Wenn wir weiter (und o.B.d.A.) voraussetzen, dass A (bzw. A' , A'') seinen Keller nie leert, gilt: Der soeben konstruierte DPDA akzeptiert/erkennt ein Eingabewort w gdw er w in einem **nicht-spontanen** Zustand akzeptiert/erkennt. \square

Satz 95

Die Klasse der deterministischen kontextfreien Sprachen (also der von DPDA's *erkannten* Sprachen) [DCFL] ist unter Komplement abgeschlossen.

Beweis:

Sei A ein DPDA, A' ein daraus wie oben beschrieben konstruierter äquivalenter DPDA. O.B.d.A. sind in A' alle Endzustände $q \in F'$ nicht spontan.

Sei $N \subseteq Q'$ die Menge aller nicht-spontanen Zustände von A' . Konstruiere den DPDA \bar{A} , indem in A' F' durch $N \setminus F'$ ersetzt wird. Dann ergibt sich aus der vorangehenden Konstruktion direkt

$$L(\bar{A}) = \overline{L(A)}.$$



Beispiel (Anwendung von Satz 95)

Korollar 96

Die Sprache

$$L = \{0^i 1^j 2^k; i = j \text{ oder } j = k\} \subset \{0, 1, 2\}^*$$

ist kontextfrei, aber nicht deterministisch kontextfrei ($\in \text{CFL} \setminus \text{DCFL}$).

Beweis:

L wird erzeugt z.B. von der CFG mit den Produktionen

$$S \rightarrow A \mid AB \mid B \mid C \mid CD \mid D \mid \epsilon$$

$$A \rightarrow 01 \mid 0A1$$

$$C \rightarrow 0 \mid 0C$$

$$B \rightarrow 2 \mid 2B$$

$$D \rightarrow 12 \mid 1D2$$

Beispiel (Anwendung von Satz 95)

Korollar 96

Die Sprache

$$L = \{0^i 1^j 2^k; i = j \text{ oder } j = k\} \subset \{0, 1, 2\}^*$$

ist kontextfrei, aber nicht deterministisch kontextfrei ($\in \text{CFL} \setminus \text{DCFL}$).

Beweis:

Wäre $L \in \text{DCFL}$, dann auch

$$L' := \bar{L} \cap 0^* 1^* 2^* = \{0^i 1^j 2^k; i \neq j \text{ und } j \neq k\}.$$

Mit Hilfe von Ogden's Lemma sieht man aber leicht, dass dies nicht der Fall ist.

4.10 $LR(k)$ -Grammatiken

Beispiel 97 (Grammatik für Arithmetische Ausdrücke)

Regeln:

$$S \rightarrow A$$

$$A \rightarrow E \mid A + E \mid A - E$$

$$E \rightarrow P \mid E * P \mid E / P$$

$$P \rightarrow (A) \mid a$$

Wir betrachten für das **Parsen** einen **bottom-up**-Ansatz, wobei die Reduktionen von links nach rechts angewendet werden.

Beispiel 97 (Grammatik für Arithmetische Ausdrücke)

Dabei können sich bei naivem Vorgehen allerdings Sackgassen ergeben, die dann aufgrund des Backtracking zu einem ineffizienten Algorithmus führen:

Regeln:

$$S \rightarrow A$$

$$A \rightarrow E \mid A + E \mid A - E$$

$$E \rightarrow P \mid E * P \mid E / P$$

$$P \rightarrow (A) \mid a$$

Ableitung:

$$a \quad + \quad a \quad * \quad a$$

$$P \quad + \quad a \quad * \quad a$$

$$E \quad + \quad a \quad * \quad a$$

$$A \quad + \quad a \quad * \quad a$$

$$S \quad + \quad a \quad * \quad a$$

Sackgasse!

Beispiel 97 (Grammatik für Arithmetische Ausdrücke)

... oder auch

Regeln:

$$S \rightarrow A$$

$$A \rightarrow E \mid A + E \mid A - E$$

$$E \rightarrow P \mid E * P \mid E / P$$

$$P \rightarrow (A) \mid a$$

Ableitung:

$$a \quad + \quad a \quad * \quad a$$

$$P \quad + \quad a \quad * \quad a$$

$$E \quad + \quad a \quad * \quad a$$

$$A \quad + \quad a \quad * \quad a$$

$$A \quad + \quad P \quad * \quad a$$

$$A \quad + \quad E \quad * \quad a$$

$$A \quad * \quad a$$

$$A \quad * \quad P$$

$$A \quad * \quad E$$

Sackgasse!

Beispiel 97 (Grammatik für Arithmetische Ausdrücke)

Zur Behebung des Problems führen wir für jede Ableitungsregel (besser hier: **Reduktionsregel**) einen **Lookahead** (der Länge k) ein (in unserem Beispiel $k = 1$) und legen fest, dass eine Ableitungsregel nur dann angewendet werden darf, wenn die nächsten k Zeichen mit den erlaubten Lookaheads übereinstimmen.

Beispiel 97 (Grammatik für Arithmetische Ausdrücke)

Produktion	Lookaheads (der Länge 1)
$S \rightarrow A$	ϵ
$A \rightarrow E$	$+, -,), \epsilon$
$A \rightarrow A + E$	$+, -,), \epsilon$
$A \rightarrow A - E$	$+, -,), \epsilon$
$E \rightarrow P$	beliebig
$E \rightarrow E * P$	beliebig
$E \rightarrow E / P$	beliebig
$P \rightarrow (A)$	beliebig
$P \rightarrow a$	beliebig

Beispiel 97 (Grammatik für Arithmetische Ausdrücke)

Damit ergibt sich

Produktion	Lookaheads
$S \rightarrow A$	ϵ
$A \rightarrow E$	$+, -,), \epsilon$
$A \rightarrow A + E$	$+, -,), \epsilon$
$A \rightarrow A - E$	$+, -,), \epsilon$
$E \rightarrow P$	beliebig
$E \rightarrow E * P$	beliebig
$E \rightarrow E / P$	beliebig
$P \rightarrow (A)$	beliebig
$P \rightarrow a$	beliebig

Ableitung:

$$\begin{array}{cccccc}
 a & + & a & * & a & \\
 P & + & a & * & a & \\
 E & + & a & * & a & \\
 A & + & a & * & a & \\
 A & + & P & * & a & \\
 A & + & E & * & a & \\
 A & + & E & * & P & \\
 & & & & E & \\
 & & & & A & \\
 & & & & S &
 \end{array}$$

Definition 98

Eine kontextfreie Grammatik ist eine $LR(k)$ -Grammatik, wenn man durch Lookaheads der Länge k erreichen kann, dass bei einer Reduktion von links nach rechts in jedem Schritt höchstens eine Produktion/Reduktion anwendbar ist.

Korollar 99

Jede kontextfreie Sprache, für die es eine $LR(k)$ -Grammatik gibt, ist deterministisch kontextfrei.

Bemerkung:

Es gibt eine (im allgemeinen nicht effiziente) Konstruktion, um aus einer $LR(k)$ -Grammatik, $k > 1$, eine äquivalente $LR(1)$ -Grammatik zu machen.

Korollar 100

Die folgenden Klassen von Sprachen sind gleich:

- *die Klasse DCFL,*
- *die Klasse der $LR(1)$ -Sprachen.*

4.11 $LL(k)$ -Grammatiken

Beispiel 101 (Noch eine Grammatik)

Regeln:

$$S \rightarrow A$$

$$A \rightarrow E \mid E + A$$

$$E \rightarrow P \mid P * E$$

$$P \rightarrow (A) \mid a$$

$$S \rightarrow A$$

$$A \rightarrow EA'$$

$$A' \rightarrow +A \mid \epsilon$$

$$E \rightarrow PE'$$

$$E' \rightarrow *E \mid \epsilon$$

$$P \rightarrow (A) \mid a$$

Wir betrachten nun für das **Parsen** einen **top-down**-Ansatz, wobei die Produktionen in Form einer Linksableitung angewendet werden.

Beispiel 101 (Noch eine Grammatik)

Regeln:

$$S \rightarrow A$$

$$A \rightarrow E \mid E + A$$

$$E \rightarrow P \mid P * E$$

$$P \rightarrow (A) \mid a$$

Ableitung:

S

A

E

P

$a + a * a$

Beispiel 101 (Noch eine Grammatik)

Wir bestimmen nun für jede Produktion $A \rightarrow \alpha$ ihre **Auswahlmenge**, das heißt die Menge aller terminalen Präfixe der Länge $\leq k$ der von α ableitbaren Zeichenreihen. Es ergibt sich (der Einfachheit lassen wir ϵ -Produktionen zu):

S	$\rightarrow A$	$\{a, (\}$
A	$\rightarrow EA'$	$\{a, (\}$
A'	$\rightarrow +A$	$\{+\}$
A'	$\rightarrow \epsilon$	$\{), \epsilon\}$
E	$\rightarrow PE'$	$\{a, (\}$
E'	$\rightarrow *E$	$\{*\}$
E'	$\rightarrow \epsilon$	$\{+,), \epsilon\}$
P	$\rightarrow (A)$	$\{(\}$
P	$\rightarrow a$	$\{a\}$

Beispiel 101 (Noch eine Grammatik)

Damit ergibt sich

S	$\rightarrow A$	$\{a, (\}$
A	$\rightarrow EA'$	$\{a, (\}$
A'	$\rightarrow +A$	$\{+\}$
A'	$\rightarrow \epsilon$	$\{), \epsilon\}$
E	$\rightarrow PE'$	$\{a, (\}$
E'	$\rightarrow *E$	$\{*\}$
E'	$\rightarrow \epsilon$	$\{+,), \epsilon\}$
P	$\rightarrow (A)$	$\{(\}$
P	$\rightarrow a$	$\{a\}$

Ableitung:

$$\begin{aligned} &S \\ &EA' \\ &aE'A' \\ &\vdots \\ &a + PE'A' \\ &\vdots \\ &a + a * PE'A' \\ &\vdots \\ &a + a * a \end{aligned}$$

Definition 102

Eine kontextfreie Grammatik ist eine $LL(k)$ -Grammatik, wenn man durch Lookaheads der Länge k erreichen kann, dass bei einer top-down Linksableitung in jedem Schritt höchstens eine Produktion anwendbar ist. Eine Sprache ist $LL(k)$, wenn es dafür eine $LL(k)$ -Grammatik gibt.

Korollar 103

Jede kontextfreie Sprache, für die es eine $LL(k)$ -Grammatik gibt, ist deterministisch kontextfrei.

Bemerkung

Man kann zeigen, dass die Klasse der $LL(k)$ -Sprachen eine echte Teilklasse der Klasse der $LL(k + 1)$ -Sprachen ist.

Bemerkungen:

- 1 Parser für $LL(k)$ -Grammatiken entsprechen der Methode des **rekursiven Abstiegs** (**recursive descent**).
- 2 $LL(k)$ ist eine strikte Teilklasse von $LR(k)$.
- 3 Es gibt $L \in \text{DCFL}$, so dass $L \notin LL(k)$ für alle k .

Bemerkung

Für die Praxis (z.B. Syntaxanalyse von Programmen) sind polynomiale Algorithmen wie CYK noch zu langsam. Für Teilklassen von CFLs sind schnellere Algorithmen bekannt, z.B.



Jay Earley:

An Efficient Context-free Parsing Algorithm.

Communications of the ACM **13**(2), pp. 94–102, 1970

4.12 Earley's Algorithmus

Sei G eine CFG, die o.B.d.A. keine ϵ -Produktion enthält (die algorithmische Behandlung des Falles $\epsilon \in L(G)$ wurde bereits besprochen) und bei der die rechte Seite einer jeden Produktion aus

$$V^+ \cup \Sigma$$

ist.

Sei $x = x_1 \cdots x_n \in \Sigma^+$ gegeben.

Definition 104

Wir definieren

$$[iAj[\alpha_1 \cdots \alpha_k \cdot \alpha_{k+1} \cdots \alpha_r,$$

falls G die Produktion

$$A \rightarrow \alpha_1 \cdots \alpha_r$$

enthält und, falls $j > i$, dann $k > 0$ und

$$\alpha_1 \cdots \alpha_k \rightarrow^* x_i \cdots x_{j-1}.$$

Wir nennen Objekte der soeben definierten Art **t-Ableitung**. (t steht dabei für **tree** oder **table** oder **top-down**.)

Earley's Algorithmus

$S_1 := \{[1S1[.\alpha; \alpha \text{ ist rechte Seite einer } S\text{-Produktion}]\}$

for $j := 1$ **to** n **do**

führe folgende Schritte so oft wie möglich aus:

if $[iAj[\alpha_1 \cdots \alpha_k.B\alpha_{k+2} \cdots \alpha_r \in S_j$ **then**

füge für jede B -Produktion $B \rightarrow \beta$ die t-Ableitung $[jBj[.\beta$ zu S_j hinzu
(falls noch nicht dort)

if $[iAj[\alpha_1 \cdots \alpha_r. \in S_j$ **then**

füge für jede t-Ableitung $[lBi[\beta_1 \cdots \beta_k.A\beta_{k+2} \cdots \beta_r$ die t-Ableitung
 $[lBj[\beta_1 \cdots \beta_kA.\beta_{k+2} \cdots \beta_r$ zu S_j hinzu

if $[jAj[.a \in S_j$ **and** $x_j = a$ **then**

füge zu S_{j+1} die t-Ableitung $[jAj + 1[a.$ hinzu

if $S_{j+1} = \emptyset$ **then return** $x \notin L$

od

if S_{n+1} enthält t-Ableitung der Form $[1Sn + 1[.\alpha.$, α rechte Seite einer S -Produktion
then return $x \in L$

Bemerkungen:

- 1 Die drei Schritte in der Lauschleife werden auch als **predictor**, **completer** und **scanner** bezeichnet.
- 2 Der Algorithmus ist eine Mischung aus einem **top-down**- und einem **bottom-up**-Ansatz.
- 3 Die Korrektheit des Algorithmus ergibt sich unmittelbar (per Induktion) aus der Definition der **t-Ableitung**.
- 4 Für eine feste CFG G und eine Eingabe x der Länge $|x| = n$ existieren höchstens $\mathcal{O}(n^2)$ t-Ableitungen.
- 5 Damit enthält jedes S_j höchstens $\mathcal{O}(n)$ t-Ableitungen.
- 6 Die erste und dritte if-Anweisung benötigen daher (pro Iteration der j -Schleife) Zeit $\mathcal{O}(n)$, die zweite if-Anweisung $\mathcal{O}(n^2)$.

Bemerkungen:

- Will man statt der ja/nein-Antwort für das Wortproblem einen (oder alle) **Ableitungsbäume**, falls $x \in L(G)$, so kann der completer-Schritt dafür geeignete Informationen kompakt abspeichern (es kann **exponentiell** viele verschiedene Ableitungsbäume geben!).

Satz 105

Die Laufzeit des Earley-Algorithmus ist, für eine feste CFG und in Abhängigkeit von der Länge n des Testworts, $\mathcal{O}(n^3)$.

Bemerkung:

Man kann zeigen (siehe [Earley's Arbeit](#)):

Ist die Grammatik eindeutig, so benötigt der completer-Schritt nur Zeit $\mathcal{O}(n)$, der ganze Algorithmus also Zeit

$$\mathcal{O}(n^2).$$

Beispiel 106

Wir betrachten wiederum unsere Grammatik für arithmetische Ausdrücke mit den Regeln:

$$S \rightarrow A$$

$$A \rightarrow E \mid A + E \mid A - E$$

$$E \rightarrow P \mid E \times P \mid E / P$$

$$P \rightarrow (A) \mid a$$

sowie das Testwort

$$a + a \times a$$

Beispiel 106

Earley's Algorithmus liefert:

S_1 : [1S1[A] [1A1[E] [1A1[A+E] [1E1[P] [1P1[a] ...

S_2 : [1P2[a] [1E2[P] [1A2[E] [1S2[A] [1A2[A+E] ...

S_3 : [1A3[A+E] [3E3[P] [3P3[a] [3E3[E×P] ...

S_4 : [3P4[a] [3E4[P] [3E4[E×P] [1A4[A+E] [1S4[A] ...

S_5 : [3E5[E×P]

S_6 : [3E6[E×P] [1A6[A+E] [1S6[A]

5. Kontextsensitive und Typ-0-Sprachen

5.1 Turingmaschinen

Turingmaschinen sind das grundlegende Modell, das wir für Computer/Rechenmaschinen verwenden. Es geht auf **Alan Turing** (1912–1954) zurück.

Definition 107

Eine **nichtdeterministische Turingmaschine** (kurz TM oder NDTM) wird durch ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ beschrieben, das folgende Bedingungen erfüllt:

- 1 Q ist eine endliche Menge von **Zuständen**.
- 2 Σ ist eine endliche Menge, das **Eingabealphabet**.
- 3 Γ ist eine endliche Menge, das **Bandalphabet**, mit $\Sigma \subseteq \Gamma$
- 4 $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R, N\})$ ist die **Übergangsfunktion**.
- 5 $q_0 \in Q$ ist der **Startzustand**.
- 6 $\square \in \Gamma \setminus \Sigma$ ist das **Leerzeichen**.
- 7 $F \subseteq Q$ ist die Menge der (**akzeptierenden**) **Endzustände**.

Eine Turingmaschine heißt **deterministisch**, falls gilt

$$|\delta(q, a)| \leq 1 \quad \text{für alle } q \in Q, a \in \Gamma.$$

Erläuterung:

Intuitiv bedeutet $\delta(q, a) = (q', b, d)$ bzw. $\delta(q, a) \ni (q', b, d)$:

Wenn sich M im Zustand q befindet und unter dem Schreib-/Lesekopf das Zeichen a steht, so geht M im nächsten Schritt in den Zustand q' über, schreibt an die Stelle des a 's das Zeichen b und bewegt danach den Schreib-/Lesekopf um eine Position nach **rechts** (falls $d = R$), **links** (falls $d = L$) bzw. lässt ihn **unverändert** (falls $d = N$).

Beispiel 108

Es soll eine TM angegeben werden, die eine gegebene Zeichenreihe aus $\{0,1\}^+$ als Binärzahl interpretiert und zu dieser Zahl 1 addiert. Folgende Vorgehensweise bietet sich an:

- 1 Gehe ganz nach rechts bis ans Ende der Zahl. Dieses Ende kann durch das erste Auftreten eines Leerzeichens gefunden werden.
- 2 Gehe wieder nach links bis zur ersten 0 und ändere diese zu einer 1. Ersetze dabei auf dem Weg alle 1en durch 0.

Also:

$$\delta(q_0, 0) = (q_0, 0, R)$$

$$\delta(q_0, 1) = (q_0, 1, R)$$

$$\delta(q_0, \square) = (q_1, \square, L)$$

$$\delta(q_1, 1) = (q_1, 0, L)$$

$$\delta(q_1, 0) = (q_f, 1, N)$$

$$\delta(q_1, \square) = (q_f, 1, N)$$

Damit ist $Q = \{q_0, q_1, q_f\}$ und $F = \{q_f\}$.

Definition 109

Eine **Konfiguration** einer Turingmaschine ist ein Tupel $(\alpha, q, \beta) \in \Gamma^* \times Q \times \Gamma^*$.

Das Wort $w = \alpha\beta$ entspricht dem Inhalt des Bandes, wobei dieses rechts und links von w mit dem Leerzeichen \square gefüllt sei. Der Schreib-/Lesekopf befindet sich auf dem ersten Zeichen von $\beta\square^\infty$.

Die **Startkonfiguration** der Turingmaschine bei Eingabe $x \in \Sigma^*$ entspricht der Konfiguration

$$(\epsilon, q_0, x),$$

d.h. auf dem Band befindet sich genau die Eingabe $x \in \Sigma^*$, der Schreib-/Lesekopf befindet sich über dem ersten Zeichen der Eingabe und die Maschine startet im Zustand q_0 .

Je nach aktuellem Bandinhalt und Richtung $d \in \{L, R, N\}$ ergibt sich bei Ausführung des Zustandsübergangs $\delta(q, \beta_1) = (q', c, d)$ folgende Änderung der Konfiguration:

$$(\alpha_1 \cdots \alpha_n, q, \beta_1 \cdots \beta_m) \rightarrow \begin{cases} (\alpha_1 \cdots \alpha_n, q', c\beta_2 \cdots \beta_m) & \text{falls } d = N, \\ & n \geq 0, m \geq 1 \\ (\epsilon, q', \square c\beta_2 \cdots \beta_m) & \text{falls } d = L, \\ & n = 0, m \geq 1 \\ (\alpha_1 \cdots \alpha_{n-1}, q', \alpha_n c\beta_2 \cdots \beta_m) & \text{falls } d = L, \\ & n \geq 1, m \geq 1 \\ (\alpha_1 \cdots \alpha_n c, q', \square) & \text{falls } d = R, \\ & n \geq 0, m = 1 \\ (\alpha_1 \cdots \alpha_n c, q', \beta_2 \cdots \beta_m) & \text{falls } d = R, \\ & n \geq 0, m \geq 2 \end{cases}$$

Der Fall $m = 0$ wird mittels $\beta_1 = \square$ abgedeckt.

Definition 110

Die von einer Turingmaschine M **akzeptierte Sprache** ist

$$L(M) = \{x \in \Sigma^*; (\epsilon, q_0, x) \rightarrow^* (\alpha, q, \beta) \text{ mit } q \in F, \alpha, \beta \in \Gamma^*\}$$

Manchmal sagen wir (in Anspielung auf ein mechanistisches Modell einer Turingmaschine), dass eine Turingmaschine in einem Zustand q_h **hält**, falls eine Konfiguration $(\alpha, q_h, c\beta)$ mit $c \in \Gamma$ erreicht wird und $\delta(q_h, c)$ nicht definiert ist. q_h kann dabei **akzeptierend** sein oder auch nicht.

5.2 Linear beschränkte Automaten

Definition 111

Eine Turingmaschine heißt **linear beschränkt** (kurz: **LBA**), falls für alle $q \in Q$ gilt:

$$(q', c, d) \in \delta(q, \square) \quad \Longrightarrow \quad c = \square.$$

Des weiteren muss die Bewegung d des Kopfes so sein, dass die vorherige Kopfbewegung wieder aufgehoben wird.

Ein Leerzeichen wird also nie durch ein anderes Zeichen überschrieben. Mit anderen Worten: Die Turingmaschine darf ausschliesslich die Positionen beschreiben, an denen zu Beginn die Eingabe x steht.

Satz 112

Die von linear beschränkten, nichtdeterministischen Turingmaschinen akzeptierten Sprachen sind genau die kontextsensitiven (also Chomsky-1) Sprachen.

Beweis:

Wir beschreiben nur die Beweisidee.

„ \implies “: Wir benutzen eine Menge von Nichtterminalsymbolen, die $Q \times \Sigma$ enthält, für die Grammatik. Die Grammatik erzeugt zunächst alle akzeptierenden Konfigurationen (der Form $\alpha q_f \beta$ mit $q_f \in F$), wobei q_f und β_1 zusammen als ein Zeichen codiert sind. Sie enthält weiterhin Regeln, die es gestatten, aus jeder Satzform, die eine Konfiguration darstellt, alle möglichen unmittelbaren Vorgängerkonfigurationen (der gleichen Länge!) abzuleiten. Die zur Anfangskonfiguration (ϵ, q_0, w) gehörige Satzform ist damit ableitbar gdw der LBA das Wort w akzeptiert.

Satz 112

Die von linear beschränkten, nichtdeterministischen Turingmaschinen akzeptierten Sprachen sind genau die kontextsensitiven (also Chomsky-1) Sprachen.

Beweis:

Wir beschreiben nur die Beweisidee.

„ \Leftarrow “: Wir simulieren mittels des LBA nichtdeterministisch und in Rückwärtsrichtung die möglichen Ableitungen der kontextsensitiven Grammatik und prüfen, ob wir die Satzform S erreichen können. Da die Grammatik längenmonoton ist, nimmt der vom LBA benötigte Platz nie zu. □

Beispiel 113

Die Sprache $L = \{a^m b^m c^m \mid m \in \mathbb{N}_0\}$ ist kontextsensitiv.

Satz 114

Das Wortproblem für LBAs bzw. für Chomsky-1-Grammatiken ist entscheidbar.

Beweis:

Siehe z.B. die Konstruktion zum vorhergehenden Satz bzw. Übungsaufgabe.

Satz 115

Die Klasse *CSL* der kontextsensitiven (bzw. Chomsky-1) Sprachen ist abgeschlossen unter den folgenden Operationen:

$$\cap, \cup, \cdot, *, -$$

Beweis:

Der Beweis für die ersten vier Operationen ergibt sich unmittelbar aus den Eigenschaften linear beschränkter Automaten, der Abschluss unter Komplement wird später gezeigt („Induktives Zählen“).

Satz 116

Folgende Probleme sind für die Klasse der Chomsky-1-Sprachen bzw. -Grammatiken nicht entscheidbar:

- 1 *Leerheit*
- 2 *Äquivalenz*
- 3 *Durchschnitt (leer, endlich, unendlich)*

Beweis:

ohne Beweis.

5.3 Chomsky-0-Sprachen

Satz 117

Zu jeder (nichtdeterministischen) TM N gibt es eine deterministische TM (DTM) M mit

$$L(N) = L(M).$$

Beweis:

Die DTM erzeugt in BFS-Manier, für $k = 0, 1, \dots$, alle Konfigurationen, die die TM N in k Schritten erreichen kann. Sie hält gdw sich dabei eine akzeptierende Konfiguration ergibt.

Satz 118

Die von (nichtdeterministischen oder deterministischen) Turingmaschinen akzeptierten Sprachen sind genau die Chomsky-0-Sprachen.

Beweis:

Wir beschreiben nur die Beweisidee.

„ \implies “: Die Grammatik erzeugt zunächst alle akzeptierenden Konfigurationen (der Form $\alpha q_f \beta$ mit $q_f \in F$). Sie enthält weiterhin Regeln, die es gestatten, aus jeder Satzform, die eine Konfiguration darstellt, alle möglichen unmittelbaren Vorgängerkonfigurationen abzuleiten. Die zur Anfangskonfiguration (ϵ, q_0, w) gehörige Satzform ist damit ableitbar gdw die TM das Wort w akzeptiert. Die Produktionen ergeben sich kanonisch aus der Übergangsrelation der TM. Sie sind i.a. nicht mehr längenmonoton!

Satz 118

Die von (nichtdeterministischen oder deterministischen) Turingmaschinen akzeptierten Sprachen sind genau die Chomsky-0-Sprachen.

Beweis:

Wir beschreiben nur die Beweisidee.

„ \Leftarrow “: Wir simulieren mit der TM alle Ableitungen der Chomsky-0-Grammatik in **BFS-Manier** und akzeptieren, falls eine solche Ableitung das Eingabewort x ergibt.

Man beachte, dass die konstruierte TM nicht unbedingt immer hält!

Eine andere Idee ist, wie im LBA-Fall die Ableitungen rückwärts zu simulieren. □

6. Übersicht Chomsky-Hierarchie

6.1 Die Chomsky-Hierarchie

Typ 3	reguläre Grammatik, rechts-/linkslineare Grammatik DFA NFA regulärer Ausdruck
DCFL	$LR(k)$ -Grammatik deterministischer Kellerautomat
Typ 2	kontextfreie Grammatik (nichtdeterministischer) Kellerautomat
Typ 1	kontextsensitive Grammatik (nichtdet.) linear beschränkter Automat (LBA)
Typ 0	Chomsky-Grammatik, Phrasenstrukturgrammatik det./nichtdet. Turingmaschine

6.2 Wortproblem

Typ 3, gegeben als DFA	lineare Laufzeit
DCFL, gegeben als DPDA	lineare Laufzeit (*)
Typ 2, CNF-Grammatik	CYK-Algorithmus, Earley-Algorithmus, Laufzeit $O(n^3)$
Typ 1	exponentiell
Typ 0	—

(*) Ein DPDA in Normalform kann noch so modifiziert werden, dass jede Folge von ϵ -Übergängen aus einer Folge von pop-Operationen gefolgt von einer Folge von push-Operationen besteht, wobei die Länge der letzteren durch eine (von $|Q|$ abhängige) Konstante beschränkt ist. Daraus folgt dann sofort, dass die Gesamtlaufzeit linear in der Eingabelänge ist.

6.3 Abschlusseigenschaften

	Schnitt	Vereinigung	Komplement	Produkt	Stern
Typ 3	ja	ja	ja	ja	ja
DCFL	nein	nein	ja	nein	nein
Typ 2	nein	ja	nein	ja	ja
Typ 1	ja	ja	ja, siehe hier	ja	ja
Typ 0	ja	ja	nein (*)	ja	ja

(*) wird im nächsten Kapitel gezeigt!

6.4 Entscheidbarkeit

	Wortproblem	Leerheit	Äquivalenz	Schnittproblem
Typ 3	ja	ja	ja	ja
DCFL	ja	ja	ja	nein (*)
Typ 2	ja	ja	nein (*)	nein
Typ 1	ja	nein (*)	nein	nein
Typ 0	nein (*)	nein	nein	nein

(*) Diese Ergebnisse werden im Abschnitt 3 des nächsten Kapitels gezeigt. Dass in jeder Spalte die darunterliegenden Einträge dann ebenfalls „nein“ sein müssen, ist klar.

Kapitel II Berechenbarkeit, Entscheidbarkeit

1. Der Begriff der Berechenbarkeit

Unsere Vorstellung ist:

$f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$ ist **berechenbar**, wenn es einen Algorithmus gibt, der f berechnet, bzw, genauer, der bei jeder Eingabe $(n_1, \dots, n_k) \in \mathbb{N}_0^k$ nach endlich vielen Schritten mit dem Ergebnis $f(n_1, \dots, n_k) \in \mathbb{N}_0$ hält.

Was bedeutet „Algorithmus“ an dieser Stelle?

AWK, B, C, Euler, Fortran, Haskell, Id, JAVA, Lisp, Modula, Oberon, Pascal, Simula, ...-Programme?

Gibt es einen Unterschied, wenn man sich auf eine bestimmte Programmiersprache beschränkt?

Analog für **partielle Funktionen**

$$f : \mathbb{N}_0^k \supseteq D \rightarrow \mathbb{N}_0$$

bedeutet **berechenbar** Folgendes:

- 1 Algorithmus hält nach endlich vielen Schritten mit dem richtigen Ergebnis, wenn $(n_1, \dots, n_k) \in D$;
- 2 hält nicht, wenn $(n_1, \dots, n_k) \notin D$.

Beispiel 119

Wir definieren folgende Funktionen:

$$f_1(n) = \begin{cases} 1 & \text{falls } n \text{ als Ziffernfolge Anfangsstück von } \pi \text{ ist} \\ 0 & \text{sonst} \end{cases}$$

$$f_2(n) = \begin{cases} 1 & \text{falls } n \text{ interpretiert als Ziffernfolge in } \pi \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$

$$f_3(n) = \begin{cases} 1 & \text{falls mindestens } n \text{ aufeinanderfolgende Ziffern} \\ & \text{in } \pi \text{ gleich 1 sind} \\ 0 & \text{sonst} \end{cases}$$

$\pi = 3, 14159265358979323846264338327950288419716939937 \dots$

Einige Beispiele sind damit:

$$f_1(314) = 1, f_1(415) = 0, f_2(415) = 1 .$$

Beispiel 119

$$f_1(n) = \begin{cases} 1 & \text{falls } n \text{ als Ziffernfolge Anfangsstück von } \pi \text{ ist} \\ 0 & \text{sonst} \end{cases}$$

Wie man leicht einsieht, ist f_1 berechenbar, denn um festzustellen, ob eine Ziffernfolge ein Anfangsstück von π ist, muss π nur auf entsprechend viele Dezimalstellen berechnet werden.

Beispiel 119

$$f_2(n) = \begin{cases} 1 & \text{falls } n \text{ interpretiert als Ziffernfolge in } \pi \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$

Für f_2 wissen wir nicht, ob es berechenbar ist. Um festzustellen, dass die Ziffernfolge in π vorkommt, müsste man π schrittweise immer genauer approximieren. Der Algorithmus würde stoppen, wenn die Ziffernfolge gefunden wird. Aber was ist, wenn die Ziffernfolge in π nicht vorkommt?

Vielleicht gibt es aber einen (noch zu findenden) mathematischen Satz, der genaue Aussagen über die in π vorkommenden Ziffernfolgen macht.

Wäre π vollkommen zufällig, was es aber nicht ist, dann würde jedes n als Ziffernfolge irgendwann vorkommen.

Beispiel 119

$$f_3(n) = \begin{cases} 1 & \text{falls mindestens } n \text{ aufeinanderfolgende Ziffern} \\ & \text{in } \pi \text{ gleich 1 sind} \\ 0 & \text{sonst} \end{cases}$$

f_3 ist berechenbar, denn $f_3 \equiv f_4$, mit

$$f_4(n) = \begin{cases} 1 & n < n_0 \\ 0 & \text{sonst} \end{cases}$$

wobei $n_0 \in \mathbb{N} \cup \{\infty\}$ die maximale Anzahl von aufeinanderfolgenden 1en in π ist. Hierbei ist es nicht von Bedeutung, wie die Zahl n_0 berechnet werden kann - wichtig ist nur, dass eine solche Zahl $n_0 \in \mathbb{N} \cup \{\infty\}$ existiert.

Hinweis:

Viele interessante Informationen zur Kreiszahl π findet man z.B. bei

- [Wikipedia](#)
- [Yasumasa Kanada's Lab](#); diese Webseite enthält auch Angaben zur Verteilung der Ziffern in der (Dezimal-/Hexadezimal)-Bruchdarstellung von π .

Weitere Vorschläge, den Begriff der Berechenbarkeit zu präzisieren und zu formalisieren:

- 1 Turing-Berechenbarkeit
- 2 Markov-Algorithmen
- 3 λ -Kalkül
- 4 μ -rekursive Funktionen
- 5 Registermaschinen
- 6 AWK, B, C, Euler, Fortran, Id, JAVA, Lisp, Modula, Oberon, Pascal, Simula, ...-Programme
- 7 while-Programme
- 8 goto-Programme
- 9 DNA-Algorithmen
- 10 Quantenalgorithmen
- 11 u.v.a.m.

Es wurde bewiesen: Alle diese Beschreibungsmethoden sind in ihrer Mächtigkeit äquivalent.

Church'sche These

Dieser formale Begriff der Berechenbarkeit stimmt mit dem intuitiven überein.

1.1 Turing-Berechenbarkeit

Definition 120

Eine (partielle) Funktion

$$f : \mathbb{N}_0^k \supseteq D \rightarrow \mathbb{N}_0$$

heißt **Turing-berechenbar**, falls es eine deterministische Turingmaschine gibt, die für jede Eingabe $(n_1, \dots, n_k) \in D$ nach endlich vielen Schritten mit dem Bandinhalt

$$f(n_1, \dots, n_k) \in \mathbb{N}_0$$

hält. Falls $(n_1, \dots, n_k) \notin D$, hält die Turingmaschine **nicht!**

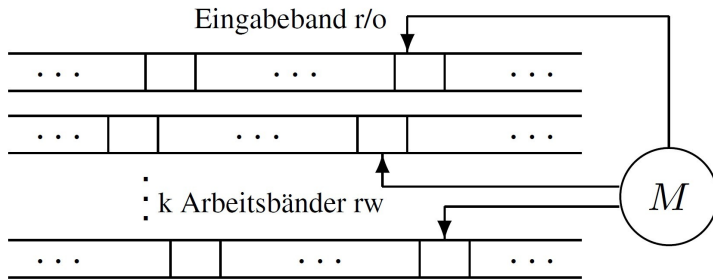
Dabei nehmen wir an, dass Tupel wie (n_1, \dots, n_k) geeignet codiert auf dem Band der Turingmaschine dargestellt werden.

Eine beliebte Modellvariante ist die k -Band-Turingmaschine, die statt einem Band k , $k \geq 1$, Arbeitsbänder zur Verfügung hat, deren Lese-/Schreibköpfe sie unabhängig voneinander bewegen kann.

Oft existiert auch ein spezielles **Eingabeband**, das nur gelesen, aber nicht geschrieben werden kann (read-only). Der Lesekopf kann jedoch normalerweise in beiden Richtungen bewegt werden.

Ebenso wird oft ein spezielles **Ausgabeband** verwendet, das nur geschrieben, aber **nicht** gelesen werden kann (write-only). Der Schreibkopf kann dabei nur nach rechts bewegt werden.

Beispiel 121 (k -Band-Turingmaschine)



Satz 122

Jede k -Band-Turingmaschine kann effektiv durch eine 1-Band-TM simuliert werden.

Beweis:

Wir simulieren die k Bänder auf k Spuren eines Bandes, wobei wir das Teilalphabet für jede Spur auch noch so erweitern, dass die Kopfposition des simulierten Bandes mit gespeichert werden kann.

Definition 123

Eine Sprache $A \subseteq \Sigma^*$ heißt **rekursiv** oder **entscheidbar**, falls es eine deterministische TM M gibt, die auf allen Eingaben $\in \Sigma^*$ hält und A erkennt.

Definition 124

Eine Sprache $A \subseteq \Sigma^*$ heißt **rekursiv aufzählbar** (r.a.) oder **semi-entscheidbar**, falls es eine TM N gibt, für die

$$L(N) = A,$$

also, falls A Chomsky-0 ist.

Definition 125

Sei $A \subseteq \Sigma^*$. Die charakteristische Funktion χ_A von A ist

$$\chi_A(w) = \begin{cases} 1 & \text{falls } w \in A \\ 0 & \text{sonst} \end{cases}$$

Definition 126

Sei $A \subseteq \Sigma^*$. χ'_A ist definiert durch

$$\chi'_A(w) = \begin{cases} 1 & \text{falls } w \in A \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Satz 127

A ist *rekursiv* $\Leftrightarrow \chi_A$ ist *berechenbar*.

Beweis:

Folgt aus der Definition von *rekursiv*: es gibt eine TM, die ja oder nein liefert.
Wandle das Ergebnis in 1 oder 0.

Satz 128

A ist rekursiv aufzählbar $\Leftrightarrow \chi'_A$ ist berechenbar.

Beweis:

Folgt unmittelbar aus der Definition.

Satz 129

A ist rekursiv $\Leftrightarrow \chi'_A$ und $\chi'_{\bar{A}}$ sind berechenbar ($\Leftrightarrow A$ und \bar{A} sind r.a.)

Beweis:

Nur \Leftarrow ist nichttrivial. Wir lassen hier eine TM für A und eine TM für \bar{A} Schritt für Schritt parallel laufen.

1.2 WHILE-Berechenbarkeit

WHILE-Programme sind wie folgt definiert:

Variablen: x_1, x_2, x_3, \dots

Konstanten: 0, 1, 2, ...

Trennsymbole: ;

Operatoren: + - \neq :=

Schlüsselwörter: WHILE DO END

Der Aufbau von WHILE-Programmen:

- $x_i := c$, $x_i := x_j + c$, $x_i := x_j - c$ sind WHILE-Programme
Die Interpretation dieser Ausdrücke erfolgt, wie üblich, mit der Einschränkung, dass $x_j - c$ als 0 gewertet wird, falls $c > x_j$.

- Sind P_1 und P_2 WHILE-Programme, so ist auch

$$P_1; P_2$$

ein WHILE-Programm.

Interpretation: Führe zuerst P_1 und dann P_2 aus.

- Ist P ein WHILE-Programm, so ist auch

$$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$$

ein WHILE-Programm.

Interpretation: Führe P solange aus, bis x_i den Wert 0 hat.

Achtung: Zuweisungen an x_i im Innern von P beeinflussen dabei den Wert von x_i !

Satz 130

Ist eine Funktion WHILE-berechenbar, so ist sie auch Turing-berechenbar.

Beweis:

Die Turingmaschine merkt sich den Programmzähler des WHILE-Programms sowie die aktuellen Werte aller Variablen. Der Platz dafür muss notfalls immer wieder angepasst werden.

Wir werden später auch die umgekehrte Aussage des obigen Satzes zeigen.

1.3 GOTO-Berechenbarkeit

GOTO-Programme sind wie folgt definiert:

Variablen: x_1, x_2, x_3, \dots

Konstanten: 0, 1, 2, ...

Trennsymbole: ;

Operatoren: + - = :=

Schlüsselwörter: IF THEN GOTO HALT

Ein GOTO-Programm ist eine Folge von markierten Anweisungen

$$M_1 : A_1; M_2 : A_2; \dots; M_k : A_k$$

wobei die A_i Anweisungen und die M_i Zielmarken für Sprünge sind.

Als Anweisungen können auftreten:

- Wertzuweisung: $x_i := x_j \pm c$
- unbedingter Sprung: GOTO M_i
- bedingter Sprung: IF $x_j = c$ THEN GOTO M_i
- Stoppanweisung: HALT

Dabei ist c jeweils eine (beliebige) Konstante.

Satz 131

Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden.

Beweis:

Ersetze jede WHILE-Schleife WHILE $x_i \neq 0$ DO P END durch folgendes Konstrukt:

```
 $M_1$ : IF  $x_i = 0$  THEN GOTO  $M_2$   
       $P$ ;  
      GOTO  $M_1$   
 $M_2$ : ...
```

Satz 132

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Beweis:

Gegeben sei das GOTO-Programm

$$M_1 : A_1; M_2 : A_2; \dots; M_k : A_k$$

Wir simulieren dies durch ein WHILE-Programm mit genau einer WHILE-Schleife:

```
c := 1;
WHILE c ≠ 0 DO
  IF c = 1 THEN A'_1 END;
  IF c = 2 THEN A'_2 END;
  ⋮
  IF c = k THEN A'_k END;
END
```

Beweis:

wobei

$$A'_i := \begin{cases} x_j := x_l \pm b; c := c + 1 & \text{falls } A_i = x_j := x_l \pm b \\ c := \ell & \text{falls } A_i = \text{GOTO } M_\ell \\ \text{IF } x_j = b \text{ THEN } c := \ell & \text{falls } A_i = \text{IF } x_j = b \text{ THEN} \\ \text{ELSE } c := c + 1 \text{ END} & \text{GOTO } M_\ell \\ c := 0 & \text{falls } A_i = \text{HALT} \end{cases}$$

Es bleibt als Übungsaufgabe überlassen, die IF-Anweisungen ebenfalls durch WHILE-Schleifen zu ersetzen. □

Satz 133

Aus Turing-Berechenbarkeit folgt GOTO-Berechenbarkeit.

Beweis:

Die Konfiguration (α, q, β) einer (det.) 1-Band-TM wird in den Variablen x_l, x_Q, x_r codiert. Die Zeichenreihen α und β werden als Zahlen (zu einer geeigneten Basis) aufgefasst, mit der niedrigstwertigen Ziffer bei der Position des Lese-/Schreibkopfes.

Jedes Tupel der Übergangsfunktion der TM wird durch ein geeignetes kleines Programmstück simuliert. Dabei werden Operationen wie Multiplikation mit, Division durch und Modularechnung zur Basis benötigt. □

1.4 Primitiv-rekursive Funktionen

Betrachte die kleinste Klasse von Funktionen $\mathbb{N}_0^k \rightarrow \mathbb{N}_0$, $k \geq 0$, für die gilt:

- 1 Sie enthält die konstanten Funktionen.
- 2 Sie enthält die Nachfolgerfunktion: $n \mapsto n + 1$.
- 3 Sie enthält die Projektionsfunktionen:

$$\text{proj}_{k,i} : \mathbb{N}_0^k \ni (x_1, \dots, x_k) \mapsto x_i \in \mathbb{N}_0$$

- 4 Sie ist abgeschlossen unter Komposition.
- 5 Sie ist abgeschlossen unter primitiver Rekursion, d.h. mit

$$g : \mathbb{N}_0^n \rightarrow \mathbb{N}_0$$

$$h : \mathbb{N}_0^{n+2} \rightarrow \mathbb{N}_0$$

ist auch

$$f(0, y_1, \dots, y_n) := g(y_1, \dots, y_n)$$

$$f(m + 1, y_1, \dots, y_n) := h(f(m, y_1, \dots, y_n), m, y_1, \dots, y_n)$$

(primitive Rekursion) in der Klasse (und sonst nichts).

Die soeben definierte Funktionenklasse sind die **primitiv-rekursiven** Funktionen.

Beispiel 134

Die folgenden Funktionen sind primitiv-rekursiv:

- 1 $(x, y) \mapsto x + y;$
- 2 $(x, y) \mapsto x * y;$
- 3 $(x, y) \mapsto \max\{x - y, 0\};$
- 4 $x \mapsto 2^x;$
- 5 $x \mapsto 2^{2^{\dots^2}}$ (Turm der Höhe x).

Satz 135

Jede primitiv-rekursive Funktion ist total.

Beweis:

Induktion über den Aufbau einer primitiv-rekursiven Funktion.

Satz 136

Jede primitiv-rekursive Funktion ist berechenbar.

Beweis:

Induktion über den Aufbau einer primitiv-rekursiven Funktion.

Korollar 137

*Die primitiv-rekursiven Funktionen sind eine **echte** Teilklasse der berechenbaren Funktionen.*

Es gibt nicht-totale berechenbare Funktionen.

Definition 138

Sei $P(x)$ ein Prädikat, d.h. ein logischer Ausdruck, der in Abhängigkeit von $x \in \mathbb{N}_0$ den Wert **true** oder **false** liefert. Dann können wir diesem Prädikat in natürlicher Weise eine 0-1 Funktion

$$\hat{P} : \mathbb{N}_0 \rightarrow \{0, 1\}$$

zuordnen, indem wir definieren, dass $\hat{P}(x) = 1$ genau dann, wenn $P(x) = \mathbf{true}$ ist.

Wir nennen $P(x)$ **primitiv-rekursiv** genau dann, wenn $\hat{P}(x)$ primitiv-rekursiv ist.

Definition 139

Beschränkter max-Operator: Zu einem Prädikat $P(x)$ definieren wir

$$q : \mathbb{N}_0 \rightarrow \mathbb{N}_0$$
$$n \mapsto \begin{cases} 0 & \text{falls } \neg P(x) \text{ für alle } x \leq n \\ 1 + \max\{x \leq n; P(x)\} & \text{sonst} \end{cases}$$

Dann gilt: Ist P primitiv-rekursiv, so auch q , denn:

$$q(0) = \hat{P}(0)$$
$$q(n+1) = \begin{cases} n+2 & \text{falls } P(n+1) \\ q(n) & \text{sonst} \end{cases}$$
$$= q(n) + \hat{P}(n+1) * (n+2 - q(n))$$

Definition 140

Beschränkter Existenzquantor: Zu einem Prädikat $P(x)$ definieren wir ein neues Prädikat $Q(x)$ mittels:

$Q(n)$ ist genau dann **true**, wenn ein $x < n$ existiert, so dass $P(x) = \mathbf{true}$.

Dann gilt: Ist P primitiv-rekursiv, so auch Q , denn:

$$\hat{Q}(0) = 0$$
$$\hat{Q}(n + 1) = \hat{P}(n) + \hat{Q}(n) - \hat{P}(n) * \hat{Q}(n)$$

Beispiel 141

Zur bijektiven Abbildung von 2-Tupeln (bzw. n -Tupeln bei iterierter Anwendung der Paarbildung) natürlicher Zahlen in die Menge der natürlichen Zahlen verwendet man eine **Paarfunktion**, z.B.:

	0	1	2	3	4	...	n_2
0	0	2	5	9	14		
1	1	4	8	13			
2	3	7	12				
3	6	11					
⋮							
n_1							

Beispiel 141

Zur bijektiven Abbildung von 2-Tupeln (bzw. n -Tupeln bei iterierter Anwendung der Paarbildung) natürlicher Zahlen in die Menge der natürlichen Zahlen verwendet man eine **Paarfunktion**, z.B.:

Betrachte: $p : \mathbb{N}_0^2 \rightarrow \mathbb{N}_0$ mit

$$p(n_1, n_2) := \frac{(n_1+n_2)(n_1+n_2+1)}{2} + n_2$$

$$c_1 : \mathbb{N}_0 \rightarrow \mathbb{N}_0$$

$$c_1(n) := s - \left(n - \frac{s(s+1)}{2}\right); \quad \text{wobei}$$

$$s := \max\left\{i; \frac{i(i+1)}{2} \leq n\right\}$$

$$c_2 : \mathbb{N}_0 \rightarrow \mathbb{N}_0$$

$$c_2(n) := n - \frac{s(s+1)}{2}, \quad s \text{ wie oben}$$

Satz 142

- 1 p stellt eine primitiv-rekursive, bijektive Paarfunktion von \mathbb{N}_0^2 nach \mathbb{N}_0 mit den Umkehrfunktionen c_1 und c_2 dar.
- 2 Die Umkehrfunktionen c_1, c_2 sind ebenfalls primitiv-rekursiv.

Beweis:

Übungsaufgabe.

1.5 LOOP-Berechenbarkeit

LOOP-Programme sind wie folgt definiert:

Variablen: x_1, x_2, x_3, \dots

Konstanten: 0, 1, 2, ...

Trennsymbole: ;

Operatoren: + - :=

Schlüsselwörter: LOOP DO END

Der Aufbau von LOOP-Programmen:

- $x_i := c$, $x_i := x_j + c$, $x_i := x_j - c$ sind LOOP-Programme.

Die Interpretation dieser Ausdrücke erfolgt, wie üblich, mit der Einschränkung, dass $x_j - c$ als 0 gewertet wird, falls $c > x_j$.

- Sind P_1 und P_2 LOOP-Programme, so ist auch

$$P_1; P_2$$

ein LOOP-Programm.

Interpretation: Führe zuerst P_1 und dann P_2 aus.

- Ist P ein LOOP-Programm, so ist auch

LOOP x_i DO P END

ein LOOP-Programm.

Interpretation: Führe P genau x_i -mal aus.

Achtung: Zuweisungen an x_i im Innern von P haben **keinen** Einfluss auf die Anzahl der Schleifendurchläufe!

Definition 143

Eine Funktion f heißt **LOOP-berechenbar** genau dann, wenn es ein LOOP-Programm gibt, das f berechnet.

LOOP-Programme können IF ... THEN ... ELSE ... END Konstrukte simulieren. Der Ausdruck IF $x = 0$ THEN A END kann durch folgendes Programm nachgebildet werden:

```
 $y := 1;$   
LOOP  $x$  DO  $y := 0$  END;  
LOOP  $y$  DO  $A$  END;
```

LOOP-berechenbare Funktionen sind immer total, denn: LOOP-Programme stoppen immer. Damit stellt sich natürlich die Frage, ob alle totalen Funktionen LOOP-berechenbar sind.

Satz 144

f ist primitiv-rekursiv \iff *f* ist LOOP-berechenbar.

Beweis:

Wir zeigen zunächst „ \Leftarrow “: Sei also P ein LOOP-Programm, das $f : \mathbb{N}_0^n \rightarrow \mathbb{N}_0$ berechnet. P verwende die Variablen x_1, \dots, x_k , $k \geq n$.

Zu zeigen: f ist primitiv-rekursiv.

Der Beweis erfolgt durch strukturelle Induktion über den Aufbau von P .

Beweis:

Induktionsanfang: $P : x_i := x_j \pm c$

Wir zeigen: Es gibt eine primitiv-rekursive Funktion

$$g_P(\underbrace{\langle a_1, \dots, a_k \rangle}_{\text{Belegung der Variablen beim Start von } P}) = \underbrace{\langle b_1, \dots, b_k \rangle}_{\text{Belegung der Variablen am Ende von } P}$$

Für $P : x_i := x_j \pm c$ erhält man:

$$g_P(\langle a_1, \dots, a_k \rangle) = \langle a_1, \dots, a_{i-1}, a_j \pm c, a_{i+1}, \dots, a_k \rangle$$

Beweis:

Induktionsschritt: Hier unterscheiden wir 2 Fälle:

- 1 Sei $P : Q; R$. Dann ist $g_P(x) = g_R(g_Q(x))$.
- 2 Sei nun $P : \text{LOOP } x_i \text{ DO } Q \text{ END}$.

Idee: Definiere Funktion $h(n, x)$, die die Belegung der Variablen berechnet, wenn man mit Belegung x startet und dann Q genau n mal ausführt. Dann ist:

$$h(0, x) = x$$

$$h(n + 1, x) = g_Q(h(n, x))$$

und damit $g_P(x) = h(d_i(x), x)$, wobei d_i die i -te Umkehrfunktion von $\langle x_1, \dots, x_k \rangle$ ist, also $d_i(\langle x_1, \dots, x_k \rangle) = x_i$.

Die Richtung „ \Rightarrow “ wird durch strukturelle Induktion über den Aufbau von f gezeigt (Übungsaufgabe). □

Definition 145

Die **Ackermann**-Funktion $a : \mathbb{N}_0^2 \rightarrow \mathbb{N}_0$:

$$a(x, y) := \begin{cases} y + 1 & \text{falls } x = 0 \\ a(x - 1, 1) & \text{falls } x \geq 1, y = 0 \\ a(x - 1, a(x, y - 1)) & \text{falls } x, y \geq 1 \end{cases}$$

Einige Eigenschaften der Ackermann-Funktion, die man per Induktion zeigen kann:

- 1 $a(1, y) = y + 2, a(2, y) = 2y + 3$ für alle y
- 2 $y < a(x, y) \quad \forall x, y$
- 3 $a(x, y) < a(x, y + 1) \quad \forall x, y$
- 4 $a(x, y + 1) \leq a(x + 1, y) \quad \forall x, y$
- 5 $a(x, y) < a(x + 1, y) \quad \forall x, y$

Genauer:

		y →					
		0	1	2	3	4	5
x	0	1	2	3	4	5	6
	1	2	3	4	5	6	7
	2	3	5	7	9	11	13
	↓ 3	5	13	29	61	125	253
	4	13	65533	$2^{65536} - 3$	$2^{2^{65536}} - 3$	$2^{2^{2^{65536}}} - 3$...
	5	65533	...				

Lemma 146

Sei P ein LOOP-Programm mit den Variablen x_1, \dots, x_k , und sei

$$f_P(n) = \max\left\{\sum_i n'_i; \sum_i n_i \leq n\right\},$$

wobei, für $i = 1, \dots, k$, n'_i der Wert von x_i nach Beendigung von P und n_i der Startwert von x_i vor Beginn von P ist. Dann gibt es ein $t \in \mathbb{N}$, so dass

$$\forall n \in \mathbb{N}_0 : f_P(n) < a(t, n).$$

Beweis:

durch Induktion über den Aufbau von P .

Induktionsanfang:

$P = x_i := x_j \pm c$, o.E. $c \in \{0, 1\}$. Es ist klar, dass

$$f_P(n) \leq 2n + 1 < 2n + 3 = a(2, n) \text{ für alle } n$$

$\Rightarrow t = 2$ tut es!

Beweis:

durch Induktion über den Aufbau von P .

Induktionsschritt:

1. Fall: $P = P_1; P_2$. Nach Induktionsannahme gibt es $k_1, k_2 \in \mathbb{N}$, so dass für $i = 1, 2$ und für alle $n \in \mathbb{N}_0$ $f_{P_i} < a(k_i, n)$.

Damit

$$f_P(n) \leq f_{P_2}(f_{P_1}(n))$$

$$\leq f_{P_2}(a(k_1, n))$$

$$< a(k_2, a(k_1, n))$$

$$\leq a(k_3, a(k_3 + 1, n))$$

$$= a(k_3 + 1, n + 1)$$

$$\leq a(k_3 + 2, n)$$

Monotonie von f_{P_2}

I.A., setze $k_3 := \max\{k_2, k_1 - 1\}$.

Monotonie!

Eigenschaft 4 der Ackermannfkt.

und $t = k_3 + 2$ tut es hier!

Beweis:

durch Induktion über den Aufbau von P .

Induktionsschritt:

2. Fall: $P = \text{LOOP } x_i \text{ DO } Q \text{ END.}$

Die Abschätzung erfolgt analog zum 1. Fall und wird als Übungsaufgabe überlassen.



Satz 147

Die Ackermann-Funktion ist nicht LOOP-berechenbar.

Beweis:

Angenommen doch. Sei P ein zugehöriges LOOP-Programm, das

$$g(n) := a(n, n)$$

berechnet.

Nach Definition von f_P gilt $g(n) \leq f_P(n)$ für alle $n \in \mathbb{N}_0$.

Wähle gemäß dem obigen Lemma t mit $f_P(\cdot) < a(t, \cdot)$ und setze $n = t$:

$$f_P(t) < a(t, t) = g(t) \leq f_P(t)$$

\Rightarrow Widerspruch!

Korollar 148

*Die primitiv-rekursiven Funktionen sind eine **echte** Teilklasse der berechenbaren totalen Funktionen.*

Beweis:

Die Ackermannfunktion ist total, berechenbar und nicht primitiv-rekursiv.

1.6 μ -rekursive Funktionen

Für eine Funktion f liefert der so genannte μ -Operator das kleinste Argument, für das f gleich 0 wird (falls ein solches existiert). Der μ -Operator gestattet so z.B., **vorab** die Anzahl der Durchläufe einer WHILE-Schleife zu bestimmen, bis die Laufvariable gleich 0 wird.

Definition 149

Sei f eine (nicht notwendigerweise totale) $k + 1$ -stellige Funktion. Die durch Anwendung des μ -Operators entstehende Funktion f_μ ist definiert durch:

$$f_\mu : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$$
$$(x_1, \dots, x_k) \mapsto \begin{cases} \min\{n \in \mathbb{N}_0; f(n, x_1, \dots, x_k) = 0\} & \text{falls} \\ & \text{dieses } n \text{ existiert und } f(m, x_1, \dots, x_k) \\ & \text{für alle } m \leq n \text{ definiert ist;} \\ \perp \text{ (undefiniert)} & \text{sonst} \end{cases}$$

Definition 150

Die Klasse der μ -rekursiven Funktionen ist die kleinste Klasse von (nicht notwendigerweise totalen) Funktionen, die die Basisfunktionen (konstante Funktionen, Nachfolgerfunktion, Projektionen) enthält und alle Funktionen, die man hieraus durch (evtl. wiederholte) Anwendung von Komposition, primitiver Rekursion und/oder des μ -Operators gewinnen kann.

Satz 151

f μ -rekursiv \iff f WHILE-berechenbar.

Beweis:

Der Beweis ist elementar.

2. Entscheidbarkeit, Halteproblem

Wir wollen nun zeigen, dass es keinen Algorithmus geben kann, der als Eingabe ein (beliebiges) Programm P und Daten x für P erhält und (für jedes solches Paar (P, x) !) **entscheidet**, ob P hält, wenn es mit Eingabe x gestartet wird.

Wir erinnern uns:

- 1 Eine Sprache A ist rekursiv gdw die charakteristische Funktion χ_A berechenbar ist.
- 2 Eine Sprache A ist rekursiv aufzählbar (r.a.) gdw die semi-charakteristische Funktion χ'_A berechenbar ist.

2.1 Rekursive Aufzählbarkeit

Definition 152

Eine Sprache $A \subseteq \Sigma^*$ heißt **rekursiv auflistbar**, falls es eine berechenbare Funktion $f : \mathbb{N}_0 \rightarrow \Sigma^*$ gibt, so dass

$$A = \{f(0), f(1), f(2), \dots\}.$$

Bemerkung: Es ist nicht verlangt, dass die Auflistung in einer gewissen Reihenfolge (z.B. lexikalisch) erfolgt!

Beispiel 153

Σ^* (mit $\Sigma = \{0, 1\}$) ist rekursiv auflistbar. Wir betrachten dazu etwa folgende Funktion:

alle nullstelligen Wörter $f(0) = \epsilon$

alle einstelligen Wörter $\begin{cases} f(1) = 0 \\ f(2) = 1 \end{cases}$

alle zweistelligen Wörter $\begin{cases} f(3) = 00 \\ f(4) = 01 \\ f(5) = 10 \\ f(6) = 11 \end{cases}$

alle dreistelligen Wörter $\begin{cases} f(7) = 000 \\ \vdots \end{cases}$

Beispiel 153

Eine weitere Möglichkeit, eine Funktion f anzugeben, die alle Wörter $\in \{0, 1\}^*$ auflistet, ist:

$$f(n) = \text{Binärkodierung von } n + 1 \text{ ohne die führende } 1$$

Also:

$$f(0) = 1\epsilon$$

$$f(1) = 10$$

$$f(2) = 11$$

$$f(3) = 100$$

$$\vdots \quad \vdots$$

Beispiel 153

$L_{TM} = \{w \in \{0, 1\}^*; w \text{ ist Codierung einer TM}\}$ ist rekursiv auflistbar:

Wir listen $\{0, 1\}^*$ rekursiv auf, prüfen jedes erzeugte Wort, ob es eine syntaktisch korrekte Codierung einer Turing-Maschine ist, und verwerfen es, falls nicht.

Wir wählen stattdessen die kanonische Auflistung von $\{0, 1\}^*$ und ersetzen jedes dabei erzeugte Wort, das keine korrekte Codierung darstellt, durch den Code einer Standard-TM, die \emptyset akzeptiert.

Satz 154

Eine Sprache A ist genau dann rekursiv auflistbar, wenn sie rekursiv aufzählbar (semi-entscheidbar) ist.

Beweis:

Wir zeigen zunächst „ \Rightarrow “.

Sei $f : \mathbb{N}_0 \rightarrow \Sigma^*$ eine berechenbare Funktion, die A auflistet.

Betrachte folgenden Algorithmus:

lies die Eingabe $w \in \Sigma^*$

$x := 0$

while true do

if $w = f(x)$ **then return** („ja“); **halt fi**

$x := x + 1$

od

Beweis:

Wir zeigen nun „ \Leftarrow “.

Sei P ein WHILE-Programm, das die semi-charakteristische Funktion χ'_A berechnet, und sei f eine berechenbare Funktion, die Σ^* auflistet.

Betrachte folgenden Algorithmus:

lies die Eingabe $n \in \mathbb{N}_0$

$count := -1; k := -1$

repeat

$k := k + 1$

$w := f(c_1(k)); m := c_2(k)$

if P hält bei Eingabe w in genau m Schritten **then**

$count := count + 1$

until $count = n$

return w

Hier sind c_1 und c_2 die Umkehrfunktionen einer Paarfunktion. □

2.2 Halteproblem

Definition 155

Unter dem **speziellen Halteproblem** H_s versteht man die folgende Sprache:

$$H_s = \{w \in \{0, 1\}^*; M_w \text{ angesetzt auf } w \text{ hält}\}$$

Hierbei ist $(M_\epsilon, M_0, M_1, \dots)$ eine berechenbare Auflistung der Turing-Maschinen.

Wir definieren weiter

Definition 156

$$L_d = \{w \in \Sigma^*; M_w \text{ akzeptiert } w \text{ nicht}\}$$

Satz 157

L_d ist nicht rekursiv aufzählbar.

Beweis:

Wäre L_d r.a., dann gäbe es ein w , so dass $L_d = L(M_w)$.

Dann gilt:

$$\begin{aligned}M_w \text{ akzeptiert } w \text{ nicht} &\Leftrightarrow w \in L_d \\ &\Leftrightarrow w \in L(M_w) \\ &\Leftrightarrow M_w \text{ akzeptiert } w\end{aligned}$$

\implies Widerspruch!

Korollar 158

L_d ist nicht entscheidbar.

Satz 159

H_s ist nicht entscheidbar.

Beweis:

Angenommen, es gäbe eine Turing-Maschine M , die H_s entscheidet. Indem man i.W. die Antworten von M umdreht, erhält man eine TM, die L_d entscheidet.

Widerspruch!

2.3 Unentscheidbarkeit

Definition 160

Unter dem (allgemeinen) Halteproblem H versteht man die Sprache

$$H = \{\langle x, w \rangle \in \{0, 1\}^*; M_x \text{ angesetzt auf } w \text{ hält}\}$$

Satz 161

Das Halteproblem H ist nicht entscheidbar.

Beweis:

Eine TM, die H entscheidet, könnten wir benutzen, um eine TM zu konstruieren, die H_s entscheidet.

Bemerkung: H und H_s sind beide rekursiv aufzählbar!

Definition 162

Seien $A, B \subseteq \Sigma^*$. Dann heißt A (effektiv) **reduzierbar auf** B gdw $\exists f : \Sigma^* \rightarrow \Sigma^*$, f total und berechenbar mit

$$(\forall w \in \Sigma^*)[w \in A \Leftrightarrow f(w) \in B].$$

Wir schreiben auch

$$A \hookrightarrow_f B \text{ bzw. } A \hookrightarrow B.$$

bzw. manchmal

$$A \leq B \text{ oder auch } A \preceq_f B.$$

Ist A mittels f auf B reduzierbar, so gilt insbesondere

$$f(A) \subseteq B \text{ und } f(\bar{A}) \subseteq \bar{B}.$$

Satz 163

Sei $A \hookrightarrow_f B$.

- (i) B rekursiv $\Rightarrow A$ rekursiv.
- (ii) B rekursiv aufzählbar $\Rightarrow A$ rekursiv aufzählbar.

Beweis:

- (i) $\chi_A = \chi_B \circ f$.
- (ii) $\chi'_A = \chi'_B \circ f$.

Definition 164

Das Halteproblem auf leerem Band H_0 ist

$$H_0 = \{w \in \{0, 1\}^*; M_w \text{ hält auf leerem Band}\}.$$

Satz 165

H_0 ist unentscheidbar (nicht rekursiv).

Beweis:

Betrachte die Abbildung f , die definiert ist durch:

$$\{0, 1\}^* \ni w \mapsto f(w),$$

$f(w)$ ist die Gödelnummer einer TM, die, auf leerem Band angesetzt, zunächst $c_2(w)$ auf das Band schreibt und sich dann wie $M_{c_1(w)}$ (angesetzt auf $c_2(w)$) verhält. Falls das Band nicht leer ist, ist es unerheblich, wie sich $M_{f(w)}$ verhält.

f ist total und berechenbar.

Es gilt: $w \in H \Leftrightarrow M_{c_1(w)}$ angesetzt auf $c_2(w)$ hält
 $\Leftrightarrow M_{f(w)}$ hält auf leerem Band
 $\Leftrightarrow f(w) \in H_0$

also $H \xrightarrow{f} H_0$ und damit H_0 unentscheidbar.

Bemerkung

Es gibt also keine allgemeine algorithmische Methode, um zu entscheiden, ob ein Programm anhält.

Satz 166 (Rice)

Sei \mathcal{R} die Menge aller (TM)-berechenbaren Funktionen und S eine nichttriviale Teilmenge von \mathcal{R} (also $S \neq \mathcal{R}$, $S \neq \emptyset$). Dann ist

$$G(S) := \{w \in \{0,1\}^*; \text{ die von } M_w \text{ berechnete Funktion ist in } S\}$$

unentscheidbar.

Beweis:

Sei Ω die total undefinierte Funktion.

1. Fall: $\Omega \in \mathcal{S}$

Sei $q \in \mathcal{R} - \mathcal{S}$ (es gibt ein derartiges q , da \mathcal{S} nichttrivial), Q eine TM für q .

Zu $w \in \{0, 1\}^*$ sei $f(w) \in \{0, 1\}^*$ Gödelnummer einer TM, für die gilt:

- 1 bei Eingabe x ignoriert $M_{f(w)}$ diese zunächst und verhält sich wie M_w auf leerem Band.
- 2 wenn obige Rechnung hält, dann verhält sich $M_{f(w)}$ wie Q auf der Eingabe x .

f ist total und berechenbar.

Beweis:

Sei Ω die total undefinierte Funktion.

1. Fall: $\Omega \in \mathcal{S}$

- $w \in H_0 \Leftrightarrow M_w$ hält auf leerem Band
- $\Leftrightarrow M_{f(w)}$ berechnet die Funktion $q (\neq \Omega)$
- \Leftrightarrow die von $M_{f(w)}$ -berechnete Funktion ist nicht in \mathcal{S}
- $\Leftrightarrow f(w) \notin G(\mathcal{S})$

Also: $\bar{H}_0 \hookrightarrow_f G(\mathcal{S})$.

H_0 unentscheidbar (nicht rekursiv) $\Rightarrow \bar{H}_0$ unentscheidbar $\Rightarrow G(\mathcal{S})$ unentscheidbar.

Wir zeigen hier nur diesen Satz. Setzt man weitere Eigenschaften von \mathcal{S} voraus, kann man sogar zeigen, dass $G(\mathcal{S})$ nicht einmal rekursiv aufzählbar ist.

Beweis:

Sei Ω die total undefinierte Funktion.

2. Fall: $\Omega \notin \mathcal{S}$

Seien $q \in \mathcal{S}, Q, f$ wie im Fall 1.

$$\begin{aligned}w \in H_0 &\Leftrightarrow M_{f(w)} \text{ berechnet die Funktion } q (\neq \Omega) \\ &\Leftrightarrow \text{die von } M_{f(w)} \text{ berechnete Funktion ist in } \mathcal{S} \\ &\Leftrightarrow f(w) \in G(\mathcal{S})\end{aligned}$$

Also: $H_0 \hookrightarrow_f G(\mathcal{S})$.

H_0 unentscheidbar $\Rightarrow G(\mathcal{S})$ unentscheidbar. □

3. Anwendung der Unentscheidbarkeitsresultate auf kontextfreie Sprachen

Wie wir gesehen haben, gilt:

- 1 die regulären Sprachen sind unter allen Booleschen Operationen abgeschlossen.
- 2 die kontextfreien Sprachen sind **nicht** unter Komplement und Durchschnitt abgeschlossen.

Können wir entscheiden, ob der Durchschnitt zweier kontextfreier Sprachen leer ist?

Sei M eine (beliebige) TM (mit nur einem Band) mit Bandalphabet Σ und Zustandsmenge Q , sei $\# \notin \Sigma \cup Q$.

Definition 167

Definiere die Sprachen

$$C_M^{(0)} := \{c_0 \# c_1^R \# c_2 \# c_3^R \dots c_m^{\pm R}; \quad m \geq 0, c_i \text{ ist Konfiguration von } M, c_0 \text{ ist Anfangskonfiguration auf leerem Band, } c_m \text{ ist Endkonfiguration, und } c_{2j+1} \text{ ist Nachfolgekonfiguration von } c_{2j} \text{ f\"ur alle } j\}$$

$$C_M^{(1)} := \{c_0 \# c_1^R \# c_2 \# c_3^R \dots c_m^{\pm R}; \quad \text{wie oben, jetzt aber: } c_{2j} \text{ ist Nachfolgekonfiguration von } c_{2j-1} \text{ f\"ur alle zutreffenden } j \geq 1\}$$

Bemerkung: Hier steht $c_m^{\pm R}$ f\"ur c_m^R , falls m ungerade ist, und f\"ur c_m sonst.

Bemerkung: $C_M^{(0)}$ enthält nicht nur „echte“ Rechnungen von M , da $c_{2j-1} \rightarrow c_{2j}$ nicht unbedingt ein Schritt sein muss; das fordern wir jeweils nur für $c_{2j} \rightarrow c_{2j+1}$.

Lemma 168

Die Sprachen $C_M^{(0)}$ und $C_M^{(1)}$ sind deterministisch kontextfrei.

Beweis:

Es ist einfach, jeweils einen DPDA dafür zu konstruieren.

Bemerkung: Ein Kellerautomat ist lange nicht so mächtig wie eine Turingmaschine. Aber **zwei** Kellerautomaten (oder eine endliche Kontrolle mit zwei Kellern) sind so mächtig wie eine Turingmaschine (siehe Übung).

Lemma 169

$$w \in H_0 \Leftrightarrow C_{M_w}^{(0)} \cap C_{M_w}^{(1)} \neq \emptyset$$

Beweis:

Unmittelbar aus der Definition der beiden Sprachen!

Bemerkung: Falls M_w deterministisch ist und $w \in H_0$, dann enthält $C_{M_w}^{(0)} \cap C_{M_w}^{(1)}$ genau ein Element, nämlich die eine Rechnung von M_w auf leerem Band.

Satz 170

Das Schnittproblem für kontextfreie Sprachen ist unentscheidbar!

Beweis:

siehe oben.

Wir haben sogar gezeigt: Das Schnittproblem für **deterministisch kontextfreie Sprachen** ist unentscheidbar!

Sei M eine (beliebige) TM (wiederum mit nur einem Band) mit Bandalphabet Σ und Zustandsmenge Q , und sei $\# \notin \Sigma \cup Q$.

Lemma 171

Die Sprache

$$\bar{C}_M := \overline{C_M^{(0)} \cap C_M^{(1)}}$$

ist kontextfrei.

Beweis:

Es ist $w \in \bar{C}_M$, falls einer der folgenden Fälle zutrifft:

- 1 w hat nicht die Form $c_0 \# c_1^R \# c_2 \# c_3^R \dots c_m^{\pm R}$;
- 2 c_0 stellt keine Anfangskonfiguration dar;
- 3 c_m stellt keine Endkonfiguration dar;
- 4 c_{i+1} ist nicht Nachfolgekonfiguration von c_i für ein i .

Für die ersten drei Fälle genügt eine reguläre Sprache, für den vierten Fall genügt die Vereinigung zweier kontextfreier Sprachen.

(Alternativ: \bar{C}_M ist die Vereinigung der Komplemente zweier deterministisch-kontextfreier Sprachen!)

Satz 172

Für eine gegebene CFG G ist es allgemein unentscheidbar, ob

$$L(G) = \Sigma^* .$$

Beweis:

Für die im vorherigen Lemma betrachtete Sprache \bar{C}_M kann eine kontextfreie Grammatik G effektiv konstruiert werden. Dann gilt:

$$L(G) = \Sigma^* \Leftrightarrow L(M) = \emptyset$$

Satz 173

Sei M eine TM, die auf jeder Eingabe mindestens zwei Schritte ausführt. Dann ist $C_M^{(0)} \cap C_M^{(1)}$ kontextfrei gdw $L(M)$ endlich ist.

Beweis:

„ \Leftarrow “ ist klar.

„ \Rightarrow “: Sei $L(M)$ unendlich. Angenommen, $C_M^{(0)} \cap C_M^{(1)}$ ist kontextfrei. Dann gibt es Wörter in $L(M)$, für die c_1 länger als die in Ogden's Lemma geforderte Konstante ist, so dass c_1 gepumpt werden kann, ohne c_0 und c_2 zu pumpen. Widerspruch!

Die Unentscheidbarkeit des Durchschnittsproblems kontextfreier Sprachen wird in der Literatur üblicherweise mit dem **Post'schen Korrespondenzproblem** (PCP) bewiesen, das nach **Emil Post** (1897–1954) benannt ist.

Definition 174 (Post'sches Korrespondenzproblem)

Gegeben: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ mit $x_i, y_i \in \Sigma^+$

Frage: gibt es eine Folge von Indizes $i_1 (= 1), i_2, \dots, i_r \in \{1, \dots, n\}$, so dass

$$x_{i_1} x_{i_2} \dots x_{i_r} = y_{i_1} y_{i_2} \dots y_{i_r} ?$$

Satz 175

PCP ist unentscheidbar.

Beweis:

Wir skizzieren, wie man mit Hilfe des PCP die Berechnung einer (det.) TM simulieren kann. Wir haben dazu (u.a.) Paare

(a, a) für alle $a \in \Sigma$

$(u_1 u_2 u_3, aqb)$ gemäß der inversen Übergangsfkt
der TM, mit $a, b \in \Sigma, q \in Q$ und
 $u_1, u_2, u_3 \in \Sigma \cup Q$

Dies bedeutet, dass die TM bei der lokalen Konfiguration aqb diese im nächsten Schritt zu $u_1 u_2 u_3$ ändert.

Beweis:

Die allgemeine Situation sieht dann so aus, dass eine geeignete Indexfolge i_1, \dots, i_k folgende Zeichenreihen erzeugt:

$$\begin{array}{l} \mathbf{x} \quad c_1 \quad \dots \quad c_{r-1} \quad x_1 \quad \dots \quad x_{i-1} \quad q \quad x_i \quad x_{i+1} \quad \dots \quad x_s \\ \mathbf{y} \quad c_1 \quad \dots \quad c_{r-1} \end{array}$$

Es müssen nun die einzelnen x_i durch Paare der Form (a, a) gematcht werden, lediglich $x_{i-1}qx_i$ kann nur durch (genau bzw. höchstens) ein Paar der zweiten Form gematcht werden.

Damit ergibt sich wieder die allgemeine Situation wie oben, mit r um 1 erhöht, und man kann das Argument per Induktion abschließen.

Wir überlassen es als Übungsaufgabe herauszufinden, wie auch Anfang (u.a. soll o.B.d.A. als erster Index $i_1 = 1$ verwendet werden) und Ende der TM-Berechnung geeignet durch das PCP simuliert werden können. □

Kapitel III Komplexität — Laufzeit und Speicherplatz

1. Notation und Grundlagen

Wir untersuchen grundlegende Eigenschaften von allgemeinen Maschinenmodellen (insbesondere der k -Band-Turingmaschine) in Bezug auf ihre Laufzeit und ihren Platzbedarf, sowie Beziehungen dieser Komplexitätsmaße beim Übergang zwischen der deterministischen (DTM) und der nichtdeterministischen (NDTM) Variante dieses Maschinenmodells.

Die k -Band-TM hat ein **read-only** Eingabeband, ein **write-only** Ausgabeband sowie k Arbeitsbänder.

Sei $L \subseteq \Sigma^*$ ein Problem/eine Sprache, und sei $w \in L$ eine **Instanz** von L . Sei weiterhin M eine TM für Problem L .

Definition 176

① M deterministisch:

$(D)TIME_M(w)$ = Anzahl der Schritte von M bei Eingabe w
(ggf ∞)

$(D)TIME_M(n) = \max\{DTIME_M(w); |w| = n\}; n \in \mathbb{N}_0$

$(D)SPACE_M(w)$ = max. Anzahl der Arbeitsbandfelder, die M
bei Eingabe w pro Arbeitsband
besucht (ggf ∞)

$(D)SPACE_M(n) = \max\{DSPACE_M(w); |w| = n\}; n \in \mathbb{N}_0$

Definition 176

② M nichtdeterministisch:

$\text{NTIME}_M(w)$ = Anzahl der Schritte einer kürzesten akzeptierenden Berechnung von M bei Eingabe w
(ggf ∞)

$\text{NTIME}_M(n) = \max\{\text{NTIME}_M(w); |w| = n\}; n \in \mathbb{N}_0$

$\text{NSPACE}_M(w)$ = Anzahl der Arbeitsbandfelder, die eine akzeptierende Berechnung von M bei Eingabe w pro Arbeitsband mindestens besucht
(ggf ∞)

$\text{NSPACE}_M(n) = \max\{\text{NSPACE}_M(w); |w| = n\}; n \in \mathbb{N}_0$

Bemerkungen:

- 1 Im nichtdeterministischen Fall gibt es auch eine **strikte** Variante der Komplexitätsmaße, die **alle** Berechnungen auf einer Eingabe zugrundelegt, nicht nur die akzeptierenden.
- 2 Der Platzbedarf einer k -Band-TM ist stets ≥ 1 ; Platzkomplexität $S(n)$ bedeutet daher eigentlich $\max\{S(n), 1\}$.
- 3 Die Laufzeit einer TM beträgt im Normalfall (andere TMs betrachten wir nicht) mindestens $n + 1$, d.h. die gesamte Eingabe wird gelesen; Zeitkomplexität $T(n)$ bedeutet daher eigentlich $\max\{T(n), n + 1\}$.

Beispiel 177

Die Sprache

$$L = \{w\#w^R; w \in \{0,1\}^*\}$$

kann jeweils von einer deterministischen TM in

- 1 Zeit $n + 1$
- 2 Platz $\log n$ (falls das Eingabeband **bidirektional** gelesen werden kann)

erkannt werden.

Definition 178

Sei $T(n)$ eine Zeitschranke, $S(n)$ eine Platzschranke. Wir definieren die folgenden Komplexitätsklassen:

- 1 $\text{DTIME}(T(n))$ ist die Klasse aller Probleme, die von einer deterministischen Turingmaschine in Zeit $T(n)$ erkannt werden können.
- 2 $\text{NTIME}(T(n))$ ist die Klasse aller Probleme, die von einer nichtdeterministischen Turingmaschine in Zeit $T(n)$ akzeptiert werden können.
- 3 $\text{DSPACE}(S(n))$ ist die Klasse aller Probleme, die von einer deterministischen Turingmaschine in Platz $S(n)$ erkannt werden können.
- 4 $\text{NSPACE}(S(n))$ ist die Klasse aller Probleme, die von einer nichtdeterministischen Turingmaschine in Platz $S(n)$ akzeptiert werden können.

2. Linearer Speed-up, lineare Bandkompression, Bandreduktion

Satz 179

Falls L von einer $S(n)$ -platzbeschränkten k -Band TM (DTM oder NDTM) erkannt bzw. akzeptiert wird, so auch von einer $k \cdot S(n)$ -platzbeschränkten 1-Band-TM.

Beweis:

Einfache Simulation!

Die folgenden Behauptungen ergeben sich durch einfache Konstruktionen unter Verwendung eines geeignet größeren Bandalphabets. Details bleiben als Übungsaufgabe überlassen.

Satz 180 (Bandkompression)

Falls L von einer $S(n)$ -platzbeschränkten k -Band-TM akzeptiert/erkannt wird, dann auch, für jedes $c > 0$, von einer $c \cdot S(n)$ -platzbeschränkten k -Band-TM.

Korollar 181

Sei $L \in \text{NSPACE}(S(n))$, $c > 0$, dann ist auch $L \in \text{NSPACE}(c \cdot S(n))$.

Satz 182 (Linearer Speed-up)

Falls L von einer $T(n)$ -zeitbeschränkten k -Band-TM akzeptiert/erkannt wird, dann auch, für jedes $c > 0$, von einer $c \cdot T(n)$ -zeitbeschränkten k -Band-TM, vorausgesetzt, dass $k \geq 2$ und $T(n) = \omega(n)$.

Korollar 183

Falls $T(n) = \omega(n)$ und $c > 0$, dann gilt

$$DTIME(T(n)) = DTIME(cT(n)).$$

Satz 184

Falls L von einer cn -zeitbeschränkten k -Band-TM akzeptiert/erkannt wird ($c \geq 1$), dann auch, für jedes $\epsilon > 0$, von einer $(1 + \epsilon)n$ -zeitbeschränkten k -Band-TM, vorausgesetzt, dass $k \geq 2$.

Definition 185 (wichtige Komplexitätsklassen)

1

$$\mathcal{P} = \bigcup_{c>0, k>0} \text{DTIME}(cn^k);$$

2

$$\mathcal{NP} = \bigcup_{c>0, k>0} \text{NTIME}(cn^k);$$

3

$$\mathcal{L} = \bigcup_{c>0} \text{DSPACE}(c \log n);$$

4

$$\mathcal{NL} = \bigcup_{c>0} \text{NSPACE}(c \log n);$$

Definition 185 (wichtige Komplexitätsklassen)

5

$$\text{PSPACE} = \bigcup_{c>0, k>0} \text{DSPACE}(cn^k);$$

6

$$\text{NPSpace} = \bigcup_{c>0, k>0} \text{NSpace}(cn^k);$$

3. Zeit und Platz

Es ist klar, dass eine $T(n)$ zeitbeschränkte k -Band-TM auch $T(n)$ -platzbeschränkt ist, da die Köpfe der k Arbeitsbänder in Zeit $T(n)$ nicht mehr Felder der Arbeitsbänder besuchen können.

Sei M eine k -Band-TM mit Bandalphabet Σ . Dann lässt sich jede **Konfiguration** von M eindeutig beschreiben durch

- 1 Angabe der Position des Lesekopfs auf dem Eingabeband (i.W. n Möglichkeiten),
- 2 Angabe der Bandkonfiguration $\alpha^{(i)}q\beta^{(i)}$, $i = 1, \dots, k$, der k Arbeitsbänder.

Die Länge einer solchen Beschreibung ist, falls M $S(n)$ -platzbeschränkt ist,

$$\leq \lceil \log n + 1 \rceil + k(\lceil \log(S(n) + 1) \rceil + S(n)) + \mathcal{O}(1).$$

Satz 186

Sei die TM M $S(n)$ -platzbeschränkt, $S(n) \geq \log n$. Dann ist die/jede kürzeste terminierende Berechnung von M $c^{S(n)}$ -zeitbeschränkt, für ein geeignetes $c > 0$.

Beweis:

Die Anzahl der **verschiedenen** Konfigurationen einer $S(n)$ -platzbeschränkten k -Band-TM, $S(n) \geq \log n$, ist $\leq c^{S(n)}$ für ein geeignetes $c > 0$, wie sich aus der vorhergehenden Bemerkung ergibt. Da sich bei einer terminierenden DTM bzw. bei einer NDTM in einem kürzesten akzeptierenden Berechnungspfad keine Konfiguration wiederholen kann, folgt daraus die Behauptung.

4. Simulation platzbeschränkter NDTMs

Satz 187 (Satz von Savitch, 1970)

Sei $L \in NSPACE(S(n))$, $S(n) \geq \log n$, $S(n)$ in Platz $(S(n))^2$ berechenbar. Dann ist
$$L \in DSPACE((S(n))^2).$$

Beweis:

Es genügt $L \in DSPACE(\mathcal{O}((S(n))^2))$ zu zeigen.

Unter den gegebenen Voraussetzungen lässt sich jede Konfiguration einer $S(n)$ -platzbeschränkten NDTM N für L in Platz $c \cdot S(n)$, für eine geeignete Konstante (abhängig von N) $c > 0$, darstellen.

Die Länge jeder kürzesten akzeptierenden Berechnung von N , die nur Platz $S(n)$ benötigt, ist beschränkt durch $2^{dS(n)}$, ebenfalls für eine geeignete Konstante $d > 0$.

Beweis (Forts.):

```
proc reach( $C_1, C_2, i$ )  
  if  $i = 0$  and  
    ( $C_1 = C_2$  or  
       $C_2$  in einem Schritt von  $N$  von  $C_1$  aus erreichbar)  
  then return true fi  
  if  $i \geq 1$  then  
    for all Konfigurationen  $C$  der Länge  $c \cdot S(n)$  do  
      if reach( $C_1, C, i - 1$ ) and reach( $C, C_2, i - 1$ ) then  
        return true  
      fi  
    fi  
  return false
```

Beweis (Forts.):

Die Prozedur **reach**, mit Parametern C, C', i , überprüft, ob N , ausgehend von der Konfiguration C , innerhalb von höchstens 2^i Schritten die Konfiguration C' erreichen kann.

Angewandt auf die Anfangskonfiguration und jede mögliche akzeptierende Endkonfiguration von N , mit drittem Parameter $dS(n)$, gestattet diese Prozedur offensichtlich, zu testen, ob N von der Anfangskonfiguration aus und mit der vorgegebenen Platzschränke eine akzeptierende Endkonfiguration erreichen kann.



5. Komplementabschluss von nichtdeterministischem Platz

Wir verwenden in diesem Abschnitt das **strikte** Komplexitätsmaß für die Platzschranke $S(n)$.

Durch die Methode des **induktiven Zählens** zeigen wir

Satz 188

Sei $S(n) \geq \log n$. Dann ist

$$NSPACE(S(n)) = co-NSPACE(S(n)).$$

Bemerkung: Für eine Komplexitätsklasse \mathcal{C} ist dabei

$$co-\mathcal{C} = \{\bar{L}; L \in \mathcal{C}\}.$$

Beweis:

Es genügt zu zeigen

$$L \in \text{NSPACE}(S(n)) \Rightarrow \bar{L} \in \text{NSPACE}(S(n)).$$

Sei $L \in \text{NSPACE}(S(n))$, N eine NDTM für L , w eine Eingabe für N , $|w| = n$. Weiter sei C_w die Menge aller ($S(n)$ -platzbeschränkten) Konfigurationen, die N bei Eingabe w erreichen kann. Es ist

$$|C_w| \leq 2^{dS(n)}$$

für eine geeignete Konstante $d > 0$.

Wir können auch annehmen, dass **jede** Berechnung von N auf w Länge τ hat, für ein $\tau < |C_w|$, und ebenso, dass N eine eindeutige akzeptierende Endkonfiguration $c^{(a)}$ besitzt. Wir modifizieren N so, dass es in der Konfiguration $c^{(a)}$ verbleibt (Endlosschleife), sobald diese Konfiguration erreicht ist.

Beweis (Forts.):

Setze

$$C_w(t) := \text{Menge der Konfigurationen, die } N \text{ nach genau} \\ t \text{ Schritten erreicht, } 0 \leq t \leq \tau \\ c_w(t) := |C_w(t)|$$

Dann gilt

$$N \text{ akzeptiert } w \text{ nicht} \Leftrightarrow c^{(a)} \notin C_w(\tau) \\ \Leftrightarrow C_w(\tau) \text{ enth\u00e4lt } c_w(\tau) \text{ von } c^{(a)} \\ \text{verschiedene Konfigurationen}$$

Wir k\u00f6nnen daher eine NDTM N' f\u00fcr \bar{L} konstruieren, indem wir nichtdeterministisch einen Beweis f\u00fcr die letzte Aussage raten und verifizieren.

Beweis (Forts.):

Die folgende Prozedur **reach** versucht, bei Eingabe t, cn, c_1, \dots, c_k , nichtdeterministisch zu beweisen, ob $c_i \in C_w(t)$ für mindestens ein $i \in \{1, \dots, k\}$. Ein Berechnungspfad, dem dies nicht gelingt, gibt „?“ zurück. cn steht dabei für $c_w(t)$.

func **reach**(t, cn, c_1, \dots, c_k)

$nc := 0$

for $c \in C_w$ **do**

nichtdeterministisch

tue nichts

oder

rate eine Berechnung von N der Länge t , mit Endkonfiguration c

if erfolgreich **then** $nc := nc + 1$ **fi**

if $c \in \{c_1, \dots, c_k\}$ **then return true fi**

od

if $nc = cn$ **then return false fi**

if $nc < cn$ **then return „?“ fi**

Beweis (Forts.):

Die Prozedur `reach` ist offensichtlich korrekt.

Der Platzbedarf von `reach` ist $\mathcal{O}(S(n))$.

Wir geben nun eine Funktion `count` an, die $c_w(t)$ induktiv berechnet (daher der Name der gesamten Beweistechnik).

Beweis (Forts.):

func count(t, cn)

$nc := 0$

for $c \in C_w$ **do**

bestimme die unmittelbaren Vorgängerkonfigurationen c_1, \dots, c_k von c

$res := \text{reach}(t - 1, cn, c_1, \dots, c_k)$

if $res = \text{true}$ **then** $nc := nc + 1$ **fi**

if $res = \text{„?“}$ **then return** „?“ **fi**

od

return nc

Wird **count** mit $cn = c_w(t - 1)$ aufgerufen und liefert es ein Ergebnis $\neq \text{„?“}$, so ist das Ergebnis gleich $c_w(t)$.

Der Platzbedarf von **count** ist $\mathcal{O}(S(n))$.

Beweis (Forts.):

Damit ergibt sich insgesamt folgende NDTM, die (in Platz $\mathcal{O}(S(n))$) genau alle $w \in \bar{L}$ akzeptiert:

Induktives Zählen:

$$c_w(0) = 1$$

for $t := 1$ **to** τ **do**

if $c_w(t - 1) \neq \text{„?“}$ **then**

$$c_w(t) := \text{count}(t, c_w(t - 1))$$

else

$$c_w(t) := \text{„?“}$$

fi

od

if $c_w(\tau) \neq \text{„?“}$ **then** $res := \text{reach}(\tau, c_w(\tau), c^{(a)})$ **fi**

if $res = \text{false}$ **then** akzeptiere w **fi**

Damit ist $\bar{L} \in \text{NSPACE}(S(n))$. □

Bemerkungen:

- Ist der Wert von $S(n)$ (und damit auch τ) unbekannt, so führt man das Verfahren nacheinander mit den Werten

$$\log n, 2 \log n, \dots$$

als Platzschränke aus. Dabei überprüft man für jeden dieser Werte, ob er zu klein ist, d.h., ob die NDTM eine Konfiguration innerhalb des gegebenen Platzes erreicht, bei der aber eine Nachfolgekonfiguration über die Platzschränke hinaus führt. Nur wenn dies der Fall ist (das kann leicht mit Hilfe der Prozedur reach überprüft werden), wird die Platzschränke weiter hochgesetzt.

- Ob auch $\text{NSPACE}(S(n))$ für $S(n) = o(\log n)$ unter Komplement abgeschlossen ist, ist ein offenes Problem (die Vermutung ist, dass eher nicht).

6. Hierarchiesätze

Satz 189

Sei $S(n)$ (bzw. $T(n)$) eine berechenbare totale Funktion. Dann gibt es eine rekursive Sprache L mit

$$L \notin DSPACE(S(n)) \text{ (bzw. } DTIME(T(n)))$$

Beweis:

Der Beweis erfolgt durch Diagonalisierung. Sei w_0, w_1, w_2, \dots eine rekursive Auflistung von $\{0, 1\}^*$, M_0, M_1, M_2, \dots eine Gödelisierung der (deterministischen) TM mit Eingabealphabet $\{0, 1\}^*$.

Beweis (Forts.):

Betrachte die Sprache

$$L = \{w_i; M_i \text{ akzeptiert } w_i \text{ nicht innerhalb Zeit } T(|w_i|)\}.$$

L ist rekursiv. Annahme, $L = L(M_i)$ für eine $T(n)$ -zeitbeschränkte (det.) TM M_i .

Falls $w_i \in L$, gilt per Definition, dass M_i w_i **nicht** in Zeit $T(|w_i|)$ akzeptiert, also Widerspruch.

Falls $w_i \notin L = L(M_i)$, akzeptiert M_i w_i nicht, also müsste per Definition $w_i \in L$ sein, also wiederum Widerspruch! □

6.1 Eine Platzhierarchie

Lemma 190

Sei L eine Sprache, die von einer $S(n)$ -platzbeschränkten (det.) TM akzeptiert wird ($S(n) \geq \log n$). Dann gibt es eine $S(n)$ -platzbeschränkte (det.) TM für L , die auf allen Eingaben hält (also L *erkennt*).

Beweis:

Man führe einen Zähler mit, der die Laufzeit in Abhängigkeit von der maximalen Anzahl von bisher besuchten Arbeitsbandfeldern beschränkt. □

Definition 191

Eine Funktion $S(n)$ heißt **platzkonstruierbar**, falls es eine $S(n)$ -platzbeschränkte DTM gibt, die bei Eingabe $w \in \{0, 1\}^n$ $S(n)$ in unärer Darstellung berechnet.

Satz 192 (Deterministischer Platzhierarchiesatz)

Sei $S_2(n) \geq \log n$ und platzkonstruierbar, sei $S_1(n) = o(S_2(n))$. Dann ist

$$DSPACE(S_1(n)) \subset DSPACE(S_2(n)).$$

Beweis:

Wir konstruieren eine DTM M , die bei Eingabe w zunächst in Platz $S_2(|w|)$ $S_2(|w|)$ Arbeitsbandfelder absteckt (und in der weiteren Rechnung nie mehr Platz benützt). M simuliert dann M_w auf Eingabe w und akzeptiert w genau dann, falls M die Simulation von M_w auf w in der vorgegebenen Platzschränke durchführen kann und M_w dabei w **nicht** akzeptiert.

Offensichtlich ist $L(M) \in \text{DSPACE}(S_2(n))$. Die Annahme $L(M) \in \text{DSPACE}(S_1(n))$ führt wie oben zu einem Widerspruch. □

6.2 Eine Zeithierarchie

Definition 193

Eine Funktion $T(n)$ heißt **zeitkonstruierbar**, falls es eine DTM gibt, die bei Eingabe $w \in \{0, 1\}^n$ nach genau $T(n)$ Schritten hält.

Ein Ergebnis wie soeben für DSPACE kann man für DTIME nicht zeigen, da für eine „real-time“-Simulation einer k -Band-TM wieder k Bänder benötigt werden, die simulierende TM aber eine fest vorgegebene Anzahl von Arbeitsbändern hat. Man kann aber zeigen:

Satz 194

$$DTIME(T(n)) \subseteq DTIME_2(T(n) \log T(n)),$$

wobei $DTIME_2(T(n))$ die auf 2-Band-DTMs in Zeit $T(n)$ erkennbaren Sprachen bezeichnet.

Satz 195 (Allgemeiner deterministischer Zeithierarchiesatz)

Sei $T_2(n)$ zeitkonstruierbar und sei $T_1(n) \log T_1(n) = o(T_2(n))$. Dann ist

$$DTIME(T_1(n)) \subset DTIME(T_2(n)).$$

Beweis:

Analog zum Platzhierarchiesatz. □

Man kann auch zeigen

Satz 196

Sei $k \geq 2$, $T_2(n)$ zeitkonstruierbar und $T_1(n) = o(T_2(n))$. Dann ist

$$DTIME_k(T_1(n)) \subset DTIME_k(T_2(n)).$$

Beweis:

[Idee] Der Beweis beruht auf einer Methode, bei der Simulation einer k -Band-DTM auf den gleichen k Bändern eine „Uhr“ für $T_2(n)$ mitlaufen zu lassen, ohne dafür einen Zeitverlust von mehr als einem konstanten Faktor in Kauf nehmen zu müssen. Dies gelingt mit Hilfe eines so genannten **distributiven Zählers**, bei dem die Darstellung der verbleibenden Zeit (also des Zählers) geschickt über das ganze Arbeitsband verteilt wird.

Beweis (Forts.):

Details dazu findet man z.B. im Buch:



Karl Rüdiger Reischuk:

Komplexitätstheorie — Band I: Grundlagen.

B.G. Teubner, Stuttgart-Leipzig, 1999

