

Informatik 3 - Klausurvorbereitung

Formale Sprachen und Berechenbarkeitstheorie

Rolf Haynberg
Universität Karlsruhe

01. März 2007

Zusammenfassung

Ursprünglich habe ich dieses Dokument zur Vorbereitung auf die Informatik 3 Klausur geschrieben. Denn das Aufschreiben in eigenen Worten half mir Sätze und Herleitungen zu verstehen bzw. zu lernen. Nun da ich dieses Dokument nochmal korrektur gelesen habe und etwas vervollständigt habe möchte ich es euch natürlich nicht vorenthalten. Beim Zusammenstellen habe ich mich insbesondere auf das Skriptum zur Vorlesung gestützt. Dieses war *Theoretische Informatik - Kurzgefasst* von *Uwe Schöning*. Wer dieses Buch schon gelesen hat wird feststellen, dass das dritte Kapitel, nämlich die Komplexitätstheorie hier komplett fehlt (Der Klausurtermin war bedrohlich nahe gerückt). Aber auch die hier vorkommenden Themen sind von mir nicht immer vollständig behandelt worden. Deswegen warne ich davor sich bei der Klausurvorbereitung nicht am Skriptum zu Vorlesung zu orientieren oder sich nur auf diese Lernhilfe zu stützen. Viel mehr soll dieses Dokument zur Erläuterung dienen.

Selbstverständlich kann ich keine Ansprüche auf die Richtigkeit der hier vorkommenden Kommentare oder Formeln erheben. Trotzdem hoffe ich, dass diese Lernhilfe dem einen oder anderen nützlich sein kann.

Viel Erfolg beim lernen,

Rolf Haynberg, 08.03.2007

Inhaltsverzeichnis

1	Formale Sprachen	3
1.1	Grammatiken	3
1.2	Reguläre Sprachen	3
1.2.1	Deterministische Endliche Automaten	3
1.2.2	Nichtdeterministische Endliche Automaten	4
1.2.3	Reguläre Ausdrücke	4
1.2.4	Das Pumping Lemma	5
1.2.5	Äquivalenzrelation	5

1.2.6	Minimalautomat	5
1.2.7	Abschlusseigenschaften	6
1.3	Kontextfreie Sprachen	6
1.3.1	Chomsky Normalform	6
1.3.2	Das Pumping Lemma	7
1.3.3	Der CYK-Algorithmus	7
1.3.4	Nichtdeterministischer Kellerautomat	8
1.3.5	Deterministischer Kellerautomat	9
1.3.6	Abschlusseigenschaften	9
1.4	Rekursive Sprachen	10
1.4.1	Turing-Maschine	10
1.4.2	Abschlusseigenschaften	11
1.5	Kontextsensitive Sprachen	11
1.5.1	Linear-Bounded-Automaton	11
1.5.2	Abschlusseigenschaften	12
2	Berechenbarkeitstheorie	12
2.1	Berechenbarkeit	12
2.1.1	Turing-Berechenbarkeit	12
2.1.2	LOOP-Programme	13
2.1.3	WHILE-Programme	13
2.1.4	GOTO-Programme	14
2.1.5	Primitiv Rekursive Funktionen	14
2.1.6	μ -rekursive Funktionen	15
2.1.7	Zusammenfassung	15
2.2	Entscheidbarkeit	16
2.2.1	Rekursiv Aufzählbar	16
2.2.2	Reduzierbarkeit	17
2.2.3	Nicht-entscheidbare Probleme	17

1 Formale Sprachen

1.1 Grammatiken

Eine Grammatik ist ein 4-Tupel:

$$G = (V, \Sigma, P, S)$$

V Menge der Variablensymbole

Σ Menge der Terminalsymbole

P Menge der Regeln oder Produktionen

S Die Startvariable $\in V$

1.2 Reguläre Sprachen

- Die Regulären Sprachen sind genau die Typ-3 Sprachen
- Alle Regeln $w_1 \rightarrow w_2 \in P$ einer Typ-3-Grammatik erfüllen folgende Bedingungen:
 - $w_1 \in V$
 - $w_2 \in \Sigma \cup \Sigma V$
 - $|w_1| \leq |w_2|$
- Die Typ-3 Sprachen können durch DFAs akzeptiert werden.

1.2.1 Deterministische Endliche Automaten

Ein Deterministischer Endlicher Automat wird durch ein 5-Tupel beschrieben:

$$M = (Z, \Sigma, \delta, z_0, E).$$

Z Menge der Zustände

Σ Eingabealphabet

δ Die Überföhrungsfunktion. $\delta : Z \times \Sigma \rightarrow Z$

z_0 Der Startzustand $\in Z$

E Menge der Endzustände $\in Z$

Endzustände werden im Graphen mit einem Doppelkreis gekennzeichnet, Startzustände mit einem eingehenden Pfeil ohne Verbindung. Übergänge werden mit Verbindungspfeilen eingezeichnet und mit dem Übergangssymbol beschriftet.

Reguläre Sprachen werden genau durch Deterministische Endlichen Automaten (DFA) akzeptiert. Dazu vergleicht man Regeln der Grammatik mit Übergängen von δ .

$\hat{\delta} : Z \times \Sigma^* \rightarrow Z$ erweitert die Definition von δ von Einzelzeichen zu Wörtern:

$$\hat{\delta}(z, \varepsilon) = z$$

$$\hat{\delta}(z, ax) = \hat{\delta}(\delta(z, a), x)$$

Wobei $a \in \Sigma$ und $x \in \Sigma^*$.

1.2.2 Nichtdeterministische Endliche Automaten

Ein Nichtdeterministischer Endlicher Automat (NFA) unterscheidet sich von einem deterministischen nur in der Definition der Übergangsfunktion δ . Sie ist bei einem NFA wie folgt definiert:

$$\delta : Z \times \Sigma \longrightarrow \mathcal{P}(Z)$$

Wobei $\mathcal{P}(Z)$ die Potenzmenge von Z ist. $\mathcal{P}(Z) = \{z \mid z \subseteq Z\}$

Das heißt bei einem NFA muss der Folgezustand nicht immer eindeutig sein. Konsequenterweise wird auch eine Menge von Startzuständen $S \subseteq Z$ zugelassen. Ein NFA akzeptiert ein Wort, wenn es (irgend)eine Ableitung gibt, die in einem Endzustand endet.

Jeder NFA lässt sich durch einen DFA darstellen. Das Verfahren zur Umformung nennt sich Teilmengenkonstruktion oder Potenzmengenkonstruktion. Dabei wird jedes Element der Potenzmenge $\mathcal{P}(Z)$ der Zustände Z eines NFA als Einzelzustand (es gibt höchstens $2^{|Z|}$ viele) eines neuen DFA angesehen.

1.2.3 Reguläre Ausdrücke

Reguläre Ausdrücke werden wie folgt, rekursiv definiert:

- \emptyset und ε sind reguläre Ausdrücke
- $a \in \Sigma$ ist ein regulärer Ausdruck
- Wenn α und β reguläre Ausdrücke sind, dann auch $\alpha\beta$, $(\alpha|\beta)$ und $(\alpha)^*$

Die Sprache, die ein regulärer Ausdruck γ beschreibt, wird wie folgt definiert:

- Falls $\gamma = \emptyset$ so ist $L(\gamma) = \emptyset$, ist $\gamma = \varepsilon$ so ist $L(\gamma) = \{\varepsilon\}$ und ist $\gamma = a$ so ist $L(\gamma) = \{a\}$
- Falls $\gamma = \alpha\beta$ so ist $L(\gamma) = L(\alpha)L(\beta)$
- Falls $\gamma = (\alpha|\beta)$ so ist $L(\gamma) = L(\alpha) \cup L(\beta)$
- Falls $\gamma = (\alpha)^*$ so ist $L(\gamma) = L(\alpha)^*$

Die Sprachen die durch reguläre Ausdrücke beschrieben werden, sind genau die regulären Sprachen. Man kann nämlich zu einem regulären Ausdruck einen entsprechenden NFA konstruieren und zu einem NFA einen regulären Ausdruck finden der die gleiche Sprache akzeptiert.

1.2.4 Das Pumping Lemma

Jede reguläre Sprache erfüllt das Pumping Lemma (jedoch nicht umgekehrt). Deswegen ist das Pumping Lemma eine Methode um festzustellen ob eine gegebene Sprache nicht regulär ist. Es lautet wie folgt:

Sei L eine reguläre Sprache. Dann gibt es eine Zahl n , so dass sich alle Wörter $x \in L$ mit $|x| \geq n$ zerlegen lassen in $x = uvw$ und dass folgende Eigenschaften erfüllt sind:

- $|v| \geq 1$
- $|uv| \leq n$
- Für alle $i = 0, 1, 2, \dots$ gilt: $uv^i w \in L$

Die Idee wird im Beweis deutlich.

1.2.5 Äquivalenzrelation

Die Nerode-Relation R_L einer Sprache L ist eine Äquivalenzrelation. Denn es gilt:

$$xR_L y \iff \forall z \in \Sigma^* : xz \in L \iff yz \in L$$

Nun gilt, dass eine Sprache L genau dann regulär ist, wenn der Index von R_L endlich ist, es also nur endlich viele Äquivalenzklassen gibt. Wählt man die Äquivalenzklassen als Zustände eines DFA, so erhält man den Minimalautomaten.

1.2.6 Minimalautomat

Der Folgende Algorithmus gibt an, welche Zustände eines gegebenen DFA noch zu verschmelzen sind, um einen Minimalautomaten zu erhalten.

1. Entferne alle Zustände, die nicht vom Startzustand zu erreichen sind.
2. Stelle eine Tabelle mit allen Zustandspaaren $\{z, z'\}, z \neq z'$ auf.
3. Markiere alle Paare $\{z, z'\}$ mit *entweder* $z \in E$ oder $z' \in E$.
4. Falls $\{\delta(z, a), \delta(z', a)\}$ für ein beliebiges $a \in \Sigma$ bereits markiert ist, markiere auch $\{z, z'\}$.
Wiederhole diesen Schritt solange, bis sich keine Änderung mehr ergibt.
5. Verschmelze alle unmarkierten Paare zu einem Zustand.

1.2.7 Abschlusseigenschaften

Die regulären Sprachen sind abgeschlossen unter:

- Vereinigung
- Schnitt
- Stern
- Produkt
- Komplement

Für reguläre Sprachen sind außerdem das Endlichkeitsproblem, das Wortproblem, das Leerheitsproblem, das Schnittproblem und das Äquivalenzproblem entscheidbar.

Beispiele:

$L = \{a^n b^n | n \geq 1\}$ ist nicht regulär.

$L_k = \{x \in (0|1)^* | \text{Die letzten } k\text{-Zeichen von } x \text{ sind } 0\}$ ist regulär.

$L_y = \{x \in \Sigma^* | \text{In } x \text{ kommt } y \in \Sigma^* \text{ vor}\}$ ist regulär.

1.3 Kontextfreie Sprachen

- Die Kontextfreien Sprachen sind genau die Typ-2 Sprachen
- Alle Regeln $w_1 \rightarrow w_2 \in P$ einer Typ-2-Grammatik erfüllen folgende Bedingungen:

- $w_1 \in V$
- $|w_1| \leq |w_2|$

1.3.1 Chomsky Normalform

Alle kontextfreien Grammatiken lassen sich in Chomsky Normalform (CNF) bringen. Darin haben alle Regeln (Produktionen) eine der beiden folgenden Formen:

$$A \rightarrow BC \text{ oder } A \rightarrow a$$

Ableitungsbäume lassen sich also immer als Binärbäume darstellen und jedes Wort w ist folglich mit $2|w| - 1$ Ableitungsschritten erzeugbar. Um eine gegebene Grammatik in CNF umzuformen kann folgender Algorithmus verwendet werden:

1. Entferne alle ε -Regeln. Dies geschieht wie folgt:
 - (a) Finde alle Variablen A mit $A \Rightarrow^* \varepsilon$.

- (b) Entferne alle Variablen der Form $A \rightarrow \varepsilon$
 - (c) Füge für jede Regel der Form $B \rightarrow xAy$ mit $A \Rightarrow^* \varepsilon$ eine neue Regel der Form $B \rightarrow xy$ hinzu.
 - (d) Ist das leere Wort in der Sprache enthalten, so benenne alle S in S' um und füge die Regel $S \rightarrow \varepsilon$ hinzu.
2. Führe für jedes Terminalsymbol eine neue Variable ein und ersetze es in allen Regeln außer in denen die bereits in Form $A \rightarrow a$ ($A \in V, a \in \Sigma$) vorliegen.
 3. Gibt es eine Regel mit mehr als zwei Variablen auf der rechten Seite ersetze jeweils zwei dieser Variablen B_1, B_2 durch eine neue Variable C und füge die Regel $C \rightarrow B_1B_2$ hinzu.
Wiederhole diesen Schritt solange wie möglich.

Es gibt auch noch die *Greibach*-Normalform in die jede kontextfreie Grammatik überführt werden kann. Dann haben alle Regeln die Form:

$$A \rightarrow aB_1B_2 \dots B_k \quad k \geq 0$$

1.3.2 Das Pumping Lemma

Jede kontextfreie Sprache erfüllt das Pumping Lemma (jedoch nicht umgekehrt). Deswegen ist das Pumping Lemma eine Methode um festzustellen ob eine gegebene Sprache nicht kontextfrei ist. Es lautet wie folgt:

Sei L eine kontextfreie Sprache. Dann gibt es eine Zahl n , so dass sich alle Wörter $z \in L$ mit $|z| \geq n$ zerlegen lassen in $x = uvwxy$ und dass folgende Eigenschaften erfüllt sind:

- $|vx| \geq 1$
- $|vwx| \leq n$
- Für alle $i = 0, 1, 2, \dots$ gilt: $uv^iwx^iy \in L$

1.3.3 Der CYK-Algorithmus

Mit Hilfe des CYK-Algorithmus kann effizient das Wortproblem für kontextfreie Sprachen gelöst werden. Im Folgenden wird w auf Enthaltensein in $L(G)$ untersucht.

1. Bringe G in CNF.
2. Zeichne eine Dreiecksmatrix mit Höhe und Breite $|w|$ und schreibe über jede Spalte nacheinander die Buchstaben von w .
3. Schreibe nun in die Zellen der erste Zeile die Variablen, aus denen das Terminalsymbol darüber abgeleitet werden kann.

4. Gehe nun von oben nach unten Zeilenweise von links nach rechts durch die Matrix. Und führe folgendes in jeder Zelle aus:
 - (a) Verwende nun zwei Marker $m1$ und $m2$. Setz $m1$ auf das oberste Kästchen der aktuellen Spalte und $m2$ auf das Kästchen rechts oberhalb der aktuellen Zelle.
 - (b) Prüfe ob es eine Produktion gibt mit $A \rightarrow BC$ wobei B das Symbol unter Marker $m1$ ist und C das Symbol unter Marker $m2$. Wenn ja, schreibe A ebenfalls in das Kästchen.
 - (c) Verschiebe $m1$ ein Kästchen tiefer und $m2$ in das nächste Kästchen rechts oberhalb des aktuell markierten Kästchens. Springe zum vorherigen Schritt wenn $m1$ noch nicht auf dem Ausgangskästchen liegt.
5. Wenn das Startsymbol in dem untersten Kästchen der ersten Zeile steht, ist das Wort in der Sprache enthalten.

1.3.4 Nichtdeterministischer Kellerautomat

Der (Nichtdeterministische) Kellerautomat, auch *Push-Down-Automaton* oder kurz PDA, ist eine Erweiterung des NFA. Er besitzt nämlich Zusätzlich einen Keller als Speicher. Eine Sprache ist genau dann kontextfrei, wenn sie von einem PDA erkannt wird.

Der PDA kann durch ein 6-Tupel beschrieben werden:

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$$

Z Menge der Zustände

Σ Eingabealphabet

Γ Kellularphabet

δ Die Überföhrungsfunktion. $\delta : Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow \mathcal{P}(Z \times \Gamma^*)$

z_0 Der Startzustand $\in Z$

$\#$ Das unterste Kellersymbol $\in \Gamma$

An der Übergangsfunktion erkennt man, den Nichtdeterminismus des PDA. Es sind nämlich mehrere Folgezustände pro Eingabe möglich. Z.B. bedeutet $\delta(z, a, A) \ni (z', B_1 B_2 B_3)$ dass, befindet der Automat sich in Zustand z , liest er das Eingabesymbol a und ist das oberste Kellersymbol A , der Automat in den Zustand z' wechseln, und dabei A durch $B_1 B_2 B_3$ im Keller ersetzen kann. Oft schreibt man der Einfachheit halber dafür auch $zaA \rightarrow z' B_1 B_2 B_3$. Eine *Konfiguration* eines Kellerautomaten ist ein Tripel $(z, x, A) \in (Z \times \Sigma^* \times \Gamma^*)$ dabei ist z der momentane Zustand, x die noch zu lesende Eingabe und A der Kellerinhalt. Außerdem sind sog. *spontane* Übergänge erlaubt. Dabei liest der Automat das Zeichen ε

Konventionell akzeptiert ein PDA ein Wort, wenn der Keller nach abarbeiten eines solchen Wortes leer ist. Jedoch könnte man jeden PDA auch so umformen, dass er via Endzustand akzeptiert. Jeder PDA kann so umgeformt werden, dass er nurnoch einen Zustand besitzt.

1.3.5 Deterministischer Kellerautomat

Ein PDA heißt *deterministisch* oder DPDA falls für alle $z \in Z, a \in \Sigma$ und $A \in \Gamma$ gilt:

$$|\delta(z, a, A)| + |\delta(z, \varepsilon, A)| \leq 1$$

Außerdem akzeptiert ein DPDA per Endzustand. Hier ist es nicht möglich dass via leerem Keller akzeptiert wird. Folglich kann man nicht jeden DPDA so umformen, dass er nurnoch einen Zustand besitzt. So wird klar, dass nicht jeder PDA durch einen DPDA simuliert werden kann!

Wie man auch sieht degenierieren Konfigurationsbäume zu Ketten (determinismus). Spontane Übergänge sind weiterhin erlaubt. Die zugehörige Sprache heißt *deterministisch kontextfrei*. Sie ist echte Obermenge der regulären Sprachen und echte Untermenge der kontextfreien Sprachen.

1.3.6 Abschlusseigenschaften

Die kontextfreien Sprachen sind abgeschlossen unter:

- Vereinigung
- Produkt
- Stern

Sie sind nicht abgeschlossen unter:

- Schnitt
- Komplement

Die deterministisch kontextfreien Sprachen sind nicht abgeschlossen unter:

- Vereinigung
- Produkt
- Stern
- Schnitt

Sondern nur unter Komplementbildung! Allerdings ist der Schnitt einer (deterministisch) kontextfreien Sprache mit einer Regulären Sprache wieder (deterministisch) kontextfrei.

Außerdem ist die Vereinigung zweier deterministisch kontextfreien Sprachen kontextfrei (aber nicht notwendigerweise deterministisch kontextfrei). Dies ist äquivalent mit folgender Aussage. Sind eine kontextfreie Sprache L_1 und eine reguläre Sprache L_2 gegeben, so ist es unentscheidbar, festzustellen, ob $L_1 = L_2$. $L = \{a^n b^n | n \geq 1\}$ ist kontextfrei.
 $L = \{a^n b^n c^m | n, m \geq 1\}$ ist kontextfrei.
 $L = \{a^n b^n c^n | n \geq 1\}$ ist nicht kontextfrei.

1.4 Rekursive Sprachen

- Die rekursiven Sprachen sind genau die Typ-0 Sprachen
- Den Regeln sind keine Einschränkungen gegeben
- Die Typ-0 Sprachen können durch Turing-Maschinen akzeptiert werden.

1.4.1 Turing-Maschine

Eine Turing-Maschine wird durch ein 7-Tupel beschrieben.

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$$

Hier bei sind:

Z Die endliche Zustandsmenge

Σ das Eingabealphabet

Γ das Arbeitsalphabet $\supset \Sigma$

δ die Übergangsfunktion $\delta : Z \times \Gamma \longrightarrow Z \times \Gamma \times \{L, R, N\}$ Dabei gibt $\{L, R, N\}$ die Bewegungsrichtung

z_0 der Startzustand

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

E die Menge der Endzustände $\subseteq Z$

Eine Konfiguration einer Turing-Maschine wird durch ein Tupel gegeben.

$$k \in \Gamma^* Z \Gamma^*$$

Der Kopf einer Turing-Maschine befindet sich unter dem ersten Zeichen das nach dem Zustand kommt.

Die Turing-Maschine akzeptiert ein Wort, wenn sie sich in endlich vielen Schritten in einem Endzustand befindet. Sie akzeptiert ein Wort nicht, indem sie entweder in einem Nicht-Endzustand oder garnicht hält. Weil die Turing-Maschine nicht halten muss, ist das Wortproblem für Typ-0 Sprachen auch nicht immer entscheidbar. Daher heißen die Typ-0 Sprachen auch *Semi-Entscheidbar*. Jedoch gibt Turing-Maschinen die bei jeder Eingabe halten (Die zugehörigen Sprachen heißen *entscheidbar*).

Nichtdeterministische allgemeine Turing-Maschinen können durch deterministische simuliert werden, weil der Berechnungsbaum einer nichtdeterministischen Turing-Maschine systematisch durchsucht werden kann.

1.4.2 Abschlusseigenschaften

Die kontextsensitiven Sprachen sind abgeschlossen unter:

- Vereinigung
- Produkt
- Stern
- Schnitt

Jedoch nicht unter Komplementbildung! Denn die Typ-0 Sprachen sind genau die Semi-Entscheidbaren Sprachen. D.h. die Funktion, die angibt ob $w \in L$, ist im Fall $w \notin L$ undefiniert. Es ist also nicht bekannt, welche Wörter nicht in L liegen und kann folglich auch kein Komplement bilden bzw. das Komplement ist undefiniert.

1.5 Kontextsensitive Sprachen

- Die kontextsensitiven Sprachen sind genau die Typ-1 Sprachen
- Alle Regeln $w_1 \rightarrow w_2 \in P$ einer Typ-1-Grammatik erfüllen die Bedingung $|w_1| \leq |w_2|$.
- Die Typ-1 Sprachen können durch LBAs akzeptiert werden.

1.5.1 Linear-Bounded-Automaton

Eine nichtdeterministische Turing-Maschine heißt linear beschränkt oder Linear-Bounded-Automaton (LBA) wenn für alle $a_1 a_2 \dots a_{n-1} a_n \in \Sigma^+$ und alle Konfigurationen $\alpha z \beta$ mit $z_0 a_1 a_2 \dots a_{n-1} \hat{a}_n \vdash^* \alpha z \beta$ gilt: $|\alpha \beta| = n$.

Dies bedeutet dass das Band auf die Länge des Eingabewortes beschränkt ist. Theoretisch könnte man ein neues Arbeitsalphabet aus m -Tupeln des ursprünglichen Arbeitsalphabet wählen. Damit hätte man die Eingabelänge künstlich um das m -fache verlängert. Aus diesem Grund spricht man von einem LBA weil die Bandlänge um einen Faktor m von der Eingabelänge abhängt. Letztendlich sind aber alle LBAs wieder überführbar in einen mit der Bandlänge gleich der Eingabelänge. Deswegen deckt die obige Definition den Begriff des LBAs vollständig ab.

Man beachte, dass die Definition von einer nichtdeterministischen Turing-Maschine ausgeht, da noch nicht geklärt ist, ob jede LBA durch eine deterministische linear beschränkte Turingmaschine (DLBA) simulierbar ist.

1.5.2 Abschlusseigenschaften

Die kontextsensitiven Sprachen sind abgeschlossen unter:

- Vereinigung
- Produkt
- Stern
- Schnitt
- Komplement

Das Leerheitsproblem und das Endlichkeitsproblem ist für kontextsensitive Sprachen unentscheidbar.

2 Berechenbarkeitstheorie

Grundlegend für dieses Themengebiet ist die Churchsche-These. Sie besagt, dass die Turing-Berechenbaren Funktionen genau mit den im intuitiven Sinne berechenbaren Funktionen übereinstimmen.

2.1 Berechenbarkeit

Die Berechenbarkeit ist auf Funktionen zugeschnitten. In der nächsten Sektion werden wir den Begriff der Entscheidbarkeit behandeln. Er ist auf Sprachen (Mengen) zugeschnitten.

2.1.1 Turing-Berechenbarkeit

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt Turing-Berechenbar, falls es eine Turing-Maschine gibt, sodass für alle $n_1, \dots, n_k, m \in \mathbb{N}$ gilt:

$$f(n_1, \dots, n_k) = m \Leftrightarrow z_0 n_1 \# n_2 \# \dots \# n_k \vdash^* \square \dots \square z_e m \square \dots \square$$

In dem Fall, dass f für eine Eingabe undefiniert ist, kann die Turing-Maschine eine unendliche Schleife gehen.

Bei Turing-Maschinen macht es keinen Unterschied von Mehrband, Einband, deterministisch oder nichtdeterministisch zu sprechen, da es alle äquivalente d.h. ineinander überführbare Modelle sind.

2.1.2 LOOP-Programme

LOOP-Programme werden induktiv wie folgt definiert:

- LOOP-Programme sind Wertzuweisungen der Form:

$$x_i := x_j + c \quad , c \in \mathbb{N}$$

- Falls P_1 und P_2 LOOP Programme sind, dann auch:

$$P_1; P_2$$

- Falls P ein LOOP-Programm ist und x_i eine Variable, dann ist auch

$$\text{LOOP } x_i \text{ DO } P \text{ END}$$

ein LOOP-Programm. Dabei wird P sooft ausgeführt wie der Wert von x_i zu Beginn angibt. P hat also keinen Einfluss auf die Anzahl der Wiederholungen.

Konventionell wird die Variable die das Endergebnis aufnimmt mit x_0 bezeichnet.

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt LOOP-Berechenbar, falls es ein LOOP-Programm gibt, das mit n_1, \dots, n_k in den Variablen x_1, \dots, x_k (und 0 in den restlichen Variablen), mit $f(n_1, \dots, n_k)$ in der Variablen x_0 endet. Es ist schnell klar, dass LOOP-Programme keine Endlosschleife gehen können. Folglich sind die durch LOOP-Programme berechenbaren Funktionen an jeder Stelle definiert. Man nennt solche Funktionen die überall Definiert sind *total*. Jedoch gilt die Umkehrung nicht! Die Ackermannfunktion ist total und berechenbar aber nicht LOOP-Berechenbar.

2.1.3 WHILE-Programme

WHILE-Programme erweitern den Begriff der LOOP-Programme um eine zusätzliche Funktion. Sie werden deshalb wie folgt induktiv definiert:

- Falls P ein LOOP-Programm ist, dann ist es auch ein WHILE-Programm
- Falls P ein WHILE-Programm ist, dann ist auch

$$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$$

ein WHILE-Programm.

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt WHILE-Berechenbar, falls es ein WHILE-Programm gibt, das mit n_1, \dots, n_k in den Variablen x_1, \dots, x_k (und 0 in den restlichen Variablen), mit $f(n_1, \dots, n_k)$ in der Variablen x_0 endet. In dem Fall, dass f für eine Eingabe undefiniert ist, geht das Programm eine Endlosschleife.

Jedes WHILE-Programm kann so umgeformt werden, dass es nur noch eine Schleife enthält.

2.1.4 GOTO-Programme

GOTO-Programme bestehen zunächst aus Marken M und Anweisungen A in der Form

$$M_1 : A_1; \dots M_n : A_n;$$

Dabei sind Anweisungen:

- Wertzuweisungen der Form: $x_i := x_j + c$
- Unbedingter Sprung: GOTO M_i
- Bedingter Sprung: IF $x_i = c$ THEN GOTO M_j
- Stopanweisung: HALT
- Sind $A_1 \dots A_n$ Anweisungen, dann auch $A_1; \dots A_n$;

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt GOTO-Berechenbar, falls es ein GOTO-Programm gibt, das mit n_1, \dots, n_k in den Variablen x_1, \dots, x_k (und 0 in den restlichen Variablen), mit $f(n_1, \dots, n_k)$ in der Variablen x_0 endet. In dem Fall, dass f für eine Eingabe undefiniert ist, geht das Programm eine Endlosschleife.

2.1.5 Primitiv Rekursive Funktionen

Primitiv rekursive Funktionen werden wie folgt induktiv definiert:

- Alle konstanten Funktionen sind primitiv rekursiv
- Alle Projektionen sind primitiv rekursiv
- Die Nachfolgerfunktion $s(n) = n + 1$ auf den natürlichen Zahlen ist primitiv rekursiv
- Die Komposition von primitiven rekursiv Funktionen ist primitiv rekursiv
- Jede Funktion die durch sog. primitive Rekursion von primitiv rekursiven Funktionen besteht ist ebenfalls primitiv rekursiv. Primitive Rekursion bedeutet dass die Definition von $f(n+1, \dots)$ zurückgeführt wird auf $h(f(n, \dots), \dots)$ und $f(0, \dots) = g(\dots)$.

2.1.6 μ -rekursive Funktionen

μ -rekursive Funktionen erweitern die Definition von primitiv rekursiven Funktionen um den sog. μ -Operator. Also:

Wenn f eine primitiv oder μ rekursive Funktion ist, dann ist $g = \mu f$ eine μ -rekursive Funktion.

Der μ -Operator ist wie folgt definiert:

$$\begin{aligned} & \mu f(n, x_1, \dots, x_n) \\ &= \min\{f(n, x_1, \dots, x_n) = 0 \text{ und f\u00fcr alle } m < n \text{ ist } f(m, x_1, \dots, x_n) \text{ definiert}\} \\ &= g(x_1, \dots, x_n) \end{aligned}$$

Es wird $\min\emptyset = \text{undefiniert}$ gesetzt.

Die primitiv rekursiven berechenbaren Funktionen sind genau die LOOP-berechenbaren Funktionen. Wir zeigen noch dass die Erweiterung der primitiv rekursiven Funktionen um dem μ -Operator genau der Erweiterung der LOOP-Programme um die WHILE-Anweisung ist. Zun\u00e4chst geben wir ein WHILE-Programm f\u00fcr $g = \mu f$ an:

```
 $x_0 := 0; y := f(0, x_1, \dots, x_n);$ 
WHILE  $y \neq 0$  DO
   $x_0 := x_0 + 1;$ 
   $y := f(x_0, x_1, \dots, x_n);$ 
END
```

Wir gehen nun von dem WHILE-Programm P: WHILE $x_i \neq 0$ DO Q END aus. Sei $h(m, x_1, \dots, x_n)$ die Funktion die den Zustand (x_1, \dots, x_n) nach m Durchl\u00e4ufen von Q wiedergibt und f die Funktion die Q berechnet. Dann ist

$$g(x_1, \dots, x_n) = h(\mu(d_i h)(x_1, \dots, x_n), f(x_1, \dots, x_n))$$

der Wert von x_1, \dots, x_n nach der minimalen Anzahl Wiederholungen von Q sodass $x_i = 0$. (Der verwendete d_i -Operator w\u00e4hlt die i -te Stelle eines Vektors aus.)

2.1.7 Zusammenfassung

Es gelten folgende \u00c4quivalenzen:

- f ist im intuitiven Sinne berechenbar
- $\Leftrightarrow f$ ist Turing-Berechenbar
- $\Leftrightarrow f$ ist WHILE-Berechenbar
- $\Leftrightarrow f$ ist GOTO-Berechenbar
- $\Leftrightarrow f$ ist eine μ -Rekursive Funktion

Weil WHILE-Programme eine Erweiterung von LOOP-Programmen sind, gilt hier keine Äquivalenz sondern nur eine Implikation:

$$f \text{ ist WHILE-Berechenbar} \Rightarrow f \text{ ist LOOP-Berechenbar}$$

Ein Beispiel dafür dass die Umkehrung nicht gilt, ist die Ackermannfunktion. Sie ist Berechenbar im intuitiven Sinne und total aber trotzdem nicht LOOP-Berechenbar.

Allerdings gilt die folgende Äquivalenz:

$$f \text{ ist LOOP-Berechenbar} \Leftrightarrow f \text{ ist primitiv rekursive Funktion}$$

2.2 Entscheidbarkeit

Der Begriff der Entscheidbarkeit ist eng verknüpft mit dem der Berechenbarkeit. Er überträgt die Berechenbarkeit auf Mengen. Dies wird in folgenden Definitionen deutlich.

Eine Menge $A \subseteq \Sigma^*$ heißt entscheidbar, genau dann, wenn eine charakteristische Funktion von A wie folgt angegeben werden kann und berechenbar ist: $\chi_A : \Sigma^* \rightarrow \{0, 1\}$,

$$\chi_A(w) = \begin{cases} 1, & w \in A \\ 0, & w \notin A \end{cases}$$

Auf einen Nenner gebracht bedeutet diese Definition χ_A berechenbar $\Leftrightarrow A$ ist entscheidbar. Diese zunächst harmlos aussehende Bedingung kann jedoch nochmals gelockert werden:

Eine Menge $A \subseteq \Sigma^*$ heißt semi-entscheidbar, genau dann, wenn eine charakteristische Funktion von A wie folgt angegeben werden kann und berechenbar ist: $\chi'_A : \Sigma^* \rightarrow \{0, 1\}$,

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

Wie man sieht muss χ'_A für den Fall $w \notin A$ nicht mehr berechenbar sein. In dem Fall, dass $w \notin A$ berechenbar ist, kann natürlich *künstlich undefiniert ausgegeben* werden. Deswegen gilt die folgende Implikation: χ_A ist berechenbar $\Rightarrow \chi'_A$ ist berechenbar bzw. A ist entscheidbar $\Rightarrow A$ ist semi-entscheidbar. Daraus folgt auch, dass eine Menge A entscheidbar ist, genau dann, wenn A und \bar{A} semi-entscheidbar sind.

2.2.1 Rekursiv Aufzählbar

Eine Menge $A \subseteq \Sigma^*$ heißt rekursiv aufzählbar, falls $A = \emptyset$ oder falls es eine totale und berechenbare Funktion $f : \mathbb{N} \rightarrow \Sigma^*$ gibt, sodass $A =$

$\{f(1), f(2), \dots\}$.

Es kann gezeigt werden, dass rekursiv aufzählbare Mengen genau die semi-entscheidbaren Mengen sind.

Es sei noch angemerkt, dass Teilmengen von rekursiv aufzählbaren Mengen nicht rekursiv aufzählbar sein müssen (denn Σ^* ist rekursiv aufzählbar und wir werden noch Mengen kennenlernen die nicht einmal semi-entscheidbar sind.)

Wir können nun eine Liste mit Äquivalenzen angeben:

- A ist semi-entscheidbar
- $\Leftrightarrow A$ ist vom Typ-0
- $\Leftrightarrow \chi'_A$ ist Turing-Berechenbar
- $\Leftrightarrow A = T(M)$ für eine Turing-Maschine M
- $\Leftrightarrow \chi'_A$ ist WHILE-Berechenbar
- $\Leftrightarrow \chi'_A$ ist GOTO-Berechenbar
- $\Leftrightarrow A$ ist rekursiv aufzählbar
- $\Leftrightarrow A$ ist Definitionsbereich einer berechenbaren Funktion
- $\Leftrightarrow A$ ist Wertebereich einer berechenbaren Funktion

2.2.2 Reduzierbarkeit

Seien A und B zwei Sprachen. A heißt reduzierbar auf B , bzw. $A \leq B$ wenn es eine totale und berechenbare Funktion f gibt mit:

$$x \in A \iff f(x) \in B$$

Nun gilt der wichtige Satz:

Falls $A \leq B$ und B (semi-)entscheidbar ist, dann ist auch A (semi-)entscheidbar. Denn $\chi_A(x) = \chi_B(f(x))$. Oft wird dieser Zusammenhang in umgekehrter Richtung verwendet. Gilt $A \leq B$ und ist A nicht entscheidbar, so ist es auch B nicht.

2.2.3 Nicht-entscheidbare Probleme

Es ist klar, dass jede Turing-Maschine durch ein Wort über dem Alphabet $\Sigma = \{0, 1\}$ angegeben werden kann. Die Turing-Maschine zu dem Wort w nennen wir von nun an M_w . Für die Wörter die keine Turing-Maschine beschreiben, ist M_w eine beliebige aber feste andere Maschine.

Das spezielle Halteproblem

Das spezielle Halteproblem ist die Menge

$$K = \{w \in \{0, 1\}^* \mid M_w \text{ angesetzt auf } w \text{ hält}\}$$

Diese Menge ist nicht entscheidbar. Denn man könnte sonst eine Turing-Maschine M angeben die K entscheidet (mit Ausgabe von 0 oder 1). Diese

Maschine M ist also auch in K da sie wie gesagt immer hält. Allerdings gibt es auch eine Turing-Maschine M' mit Codewort w' die nicht hält wenn M das Wort w' akzeptiert und hält wenn M das Codewort w' nicht akzeptiert. Folglich wäre unsere gewählte Maschine M nicht in der Lage K richtig zu entscheiden. Weil aber die Wahl von M frei war, gibt es keine Turing-Maschine die K entscheidet.

Das allgemeine Halteproblem

Das allgemeine Halteproblem ist die Menge

$$H = \{w\#x \in \{0,1\}^* \mid M_w \text{ angesetzt auf } x \text{ hält}\}$$

Wir geben eine totale und berechenbare Funktion f an, sodass $x \in K \Leftrightarrow f(x) \in H$. $f = x\#x$. Jetzt ist die Äquivalenz klar und es gilt $K \leq H$. Da K nicht entscheidbar ist, ist auch H nicht entscheidbar.

Das Halteproblem auf leerem Band

Das Halteproblem auf leerem Band ist die Menge

$$H_0 = \{w \in \{0,1\}^* \mid M_w \text{ angesetzt auf leeres Band hält}\}$$

Auch hier geben wir eine Funktion f an, sodass $x \in K \Leftrightarrow f(x) \in H_0$. $f(x)$ schreibt zunächst x auf das Band und verhält sich dann wie M_x . Auch hier ist die Äquivalenz klar und es gilt $K \leq H$. Da K nicht entscheidbar ist, ist auch H_0 nicht entscheidbar.

Satz von Rice

Sei \mathcal{R} die Menge aller Turing-Berechenbaren Funktionen und \mathcal{S} eine beliebige Teilmenge davon (außer $\mathcal{S} = \emptyset$ oder $\mathcal{S} = \mathcal{R}$). Dann ist die Menge

$$C(\mathcal{S} = \{w \mid \text{Die von } M_w \text{ berechnete Funktion liegt in } \mathcal{S}\})$$

unentscheidbar.

D.h. einer Turing-Maschine kann algorithmisch nicht angesehen werden, welche Funktion sie berechnet.

Das Postsche Korrespondenzproblem

Das Problem wird mit PCP abgekürzt.

gegeben: Endliche Folge von Wortpaaren $(x_1, y_1), \dots, (x_k, y_k)$ mit $x_i \in \Sigma^+$.

gesucht: Eine Folge von Indizes $i_1, \dots, i_n \in \{1, \dots, k\}$ mit $x_{i_1} \dots x_{i_n} = y_{i_1} \dots y_{i_n}$.

Eine Abwandlung des PCP ist das MPCP (Modified Post's Correspondence Problem). Dabei wird vorausgesetzt das $i_1 = 1$.

Es lässt sich zeigen, dass $H \leq MPCP \leq PCP$. Damit ist das PCP also unentscheidbar.

Es gibt offensichtlich auch einen Algorithmus, der Indexfolgen systematisch auf eine Lösung prüft. Folglich ist das PCP rekursiv aufzählbar und

damit semi-entscheidbar. Alle oben genannten unentscheidbaren Probleme können auf das PCP zurückgeführt werden und sind somit auch alle semi-entscheidbar.

Unentscheidbare Probleme bei deterministisch kontextfreien Sprachen

Seien G_1, G_2 zwei deterministisch kontextfreien Grammatiken. Dann sind folgende Fragestellungen unentscheidbar:

- Ist $L(G_1) \cap L(G_2) = \emptyset$?
- Ist $|L(G_1) \cap L(G_2)| = \infty$?
- Ist $L(G_1) \cap L(G_2)$ kontextfrei ?
- Ist $L(G_1) \subseteq L(G_2)$?

Für den Beweis geben wir zwei deterministisch kontextfreie Grammatiken G_1, G_2 und ein Korrespondenzproblem K an. Und zwar derart, dass $L(G_1) \cap L(G_2)$ nicht leer ist, genau dann, wenn K eine Lösung besitzt. Damit ist das Korrespondenzproblem auf das Schnittproblem für deterministisch kontextfreie Grammatiken reduziert worden. Weil das Korrespondenzproblem unentscheidbar ist, ist es auch das o.g. Schnittproblem. Für die Ausformulierungen der Grammatiken siehe Schöning, S. 137.

Außerdem kann die Lösungsfolge des Korrespondenzproblems beliebig oft wiederholt werden und besitzt damit unendlich viele Lösungen bzw. der Schnitt unendlich viele Wörter. Somit ist auch dieses Problem unentscheidbar.

Wenn der Schnitt der Sprachen nicht leer ist, enthält Wörter der Form uv^Rv^R (so wurden die Grammatiken formuliert) und wäre aufgrund des Pumping-Lemmas nicht kontextfrei. Es ist also auch die dritte Frage unentscheidbar.

Weil die Sprachen deterministisch kontextfrei sind, kann man effektiv die Komplemente bilden. Nun ist

$$\begin{aligned} L_1 \cap L_2 = \emptyset &\Leftrightarrow L_1 \subseteq \bar{L}_2 \\ &\Leftrightarrow L_1 \cup \bar{L}_2 = \bar{L}_2 \\ &\Leftrightarrow L_3 = \bar{L}_2 \end{aligned}$$

Wegen den Äquivalenzen und damit der Reduktion des PCPs auf das Inklusionsproblem, ist dieses auch unentscheidbar.

Es sei jedoch angemerkt, dass L_3 nicht notwendigerweise deterministisch ist. Das Äquivalenzproblem ($L(G_1) = L(G_2)$?) ist also für kontextfreie Sprachen unentscheidbar aber nicht notwendigerweise für deterministisch kontextfreie Sprachen. Tatsächlich ist es für diese sogar entscheidbar!

Weiter folgt auch die Unentscheidbarkeit des Äquivalenzproblems für jeden Formalismus in den kontextfreie Grammatiken überführt werden können: nichtdeterministische Kellerautomaten, BNF, EBNF, Syntaxdiagramme, LBAs, kontextsensitive Grammatiken, Turingmaschinen, LOOP-, WHILE-, GOTO-Programme usw.!

Unentscheidbare Probleme bei kontextfreien Sprachen Seien G und H kontextfreie Grammatiken. Dann sind folgende Fragestellungen unentscheidbar:

- Ist $L(G) = L(H)$?
- Ist G mehrdeutig?
- Ist $\overline{L(G)}$ kontextfrei?
- Ist $L(G)$ regulär?
- Ist $L(G)$ deterministisch kontextfrei?

Dass die erste Fragestellung unentscheidbar ist, wurde bereits im vorherigen Paragraphen geklärt.

Für die weiteren Beweise benötigen wir wieder G_1 , G_2 und K aus dem vorherigen Paragraphen. Sei nun $L(G_3) = L(G_1) \cap L(G_2)$ eine kontextfreie Sprache. Jetzt ist klar, dass K genau dann eine Lösung hat, wenn es ein Wort in $L(G_3)$ gibt mit zwei Ableitungsregeln (die eine kommt aus den G_1 -Regeln die andere aus den G_2 -Regeln). Daher ist das Mehrdeutigkeitsproblem unentscheidbar.

Weil G_1, G_2 deterministisch kontextfrei sind, können wir auch die Komplemente G'_1 und G'_2 bilden. Die Vereinigung $L(G_4) = L(G'_1) \cup L(G'_2)$ ist kontextfrei (aber nicht notwendigerweise deterministisch kontextfrei). Nach dem oben bewiesen ist es nun klar: $\overline{L(G_4)}$ ist kontextfrei $\Leftrightarrow \overline{L_1 \cup L_2} = L_1 \cap L_2$ ist kontextfrei.

Die Art und Weise mit der das Korrespondenzproblem reduziert wurde (siehe oben) führt zu folgender Äquivalenz: K besitzt eine Lösung $\Leftrightarrow L_1 \cap L_2 \neq \emptyset \Leftrightarrow L(G_4) \neq \Sigma^*$. Letzteres ist eine reguläre Sprache. Also ist auch diese dritte Fragestellung unentscheidbar genau wie ihr Formalismus, die deterministischen kontextfreien Sprachen, womit die vierte Fragestellung geklärt ist. Kontextfrei ist $L(G_4)$ ja nach Konstruktion, was somit schon entscheidbar ist.