Lernzusammenfassung der Vorlesung Informatik IV als Vorbereitung auf die Informatik B Vordiplomprüfung bei Prof. Dr. Baier

Version vom 24. August 2001

Hallo!
Dies ist eine überarbeitete Fassung der Lernsammlung für die Informatik IV Vorlesung. Sie soll vor allen Dingen dazu dienen, sich auf die Informatik B Prüfung vorzubereiten. Der Informatik III Teil ist nicht abgedeckt, ist aber für die Prüfung auch zu lernen. Mittlerweile habe ich von einigen Kommilitonen mehr oder weniger konstruktive Kritik bekommen und habe versucht diese in dieses Skript mit hineinzuarbeiten. Desweiteren habe ich das Skript selbst noch einmal korrekturgelesen und überarbeitet. Ich kann aber selbstverständlich wieder nicht für Richtigkeit und Fehlerfreiheit garantieren. Über weitere Verbesserungsvorschläge freue ich mich sehr. Hier noch einmal meine Emailadresse: *ctornau@yahoo.de*. Markus Lucht hat an der überarbeiteten Fassung nicht mehr mitgeschrieben, da er die Prüfung noch nicht machen wird. Ihm gebührt jedoch weiterhin mein Dank, da er viele Teile dieses Skriptes geschrieben hat. Ich wünsche Euch weiter viel Erfolg beim Lernen und vor allen Dingen auch eine gute Note!

Christoph Tornau

Inhaltsverzeichnis

1 RM - Registermaschinen (Random Access Machine)

1.1 Definition

Eine **Registermaschine** ist ein rechnernahes abstraktes Rechnermodell. Registermaschinen verfügen über unendlich viele **Register** (Speicher). In diesen können **entweder Variablen oder das Programm** abgelegt werden. Ein **Befehlszähler** zeigt auf die aktuelle Position im Programm, wo gearbeitet wird. Über **Marken** im Programm und **Sprungbefehle** läßt sich der Befehlszähler ändern, so daß innerhalb des Programms meistens auch bedingt, gesprungen werden kann. Gibt es keinen expliziten Befehl zur Änderung der Position, so wird mit jedem Befehl der Befehlszähler um einen Befehl nach vorne gesetzt. Eigentliche Berechungen finden nicht auf den Registern statt, sondern dazu werden die Variablen in den Registerzellen in einen **Akkumulator** übertragen und dort verarbeitet.

1.2 Syntax

Als **Operandentypen** der Befehle sind 3 verschiedene Typen zulässig (Der Akkumulator wird auch mit $c(0)$ bezeichnet:

- Konstanten $\#k$: $c(0) = k$
- direkte Adressierung i: $c(0) = c(i)$ Inhalt von Registerzelle i
- indirekte Adressierung $*i$: $c(0) = c(c(i))$ Inhalt der Registerzelle $c(i)$

In Informatik IV wurde immer eine Ein-Adress-Registermaschine verwendet.

arithmetische Operationen: $ADD, MULT, SUB$ und DIV
Sprungbefehle: bedingte und unbedingte Sprünge sind möglich. Für Vergleiche stehen die üblichen Operatoren zur Verfügung ($>$, $\geq$, $<$, $\leq$, $=$, $\neq$)
Terminierung: sobald der Befehlszähler ∞ ist, oder END aufgerufen wird. (unzulässige Operationen führen auch zum Abbruch)
Darstellung ganzer Zahlen: benutze 2 Register (1. für's VZ, 2. für den Wert)
Darstellung rat. Zahlen: benutze 3 Register (1. für's VZ, 2. für Vorkommateil 3. für Nachkommateil)

1.3 Definition: Durch Registermaschine berech. partielle Funktion

Die durch $\mathcal{R}$ berechnete partielle Funktion

$$f_{\mathcal{R}} : \mathbb{N}^k \to \mathbb{N}$$

ist gegeben durch:

- $f_{\mathcal{R}}(n_1, n_2, ...n_k) = \perp$, falls $\mathcal{R}$ für die initiale Registerbelegung c$[n_1, ..., n_k]$ nicht terminiert.
- $f_{\mathcal{R}}(n_1, n_2, ...n_k) = c(i)$, falls $\mathcal{R}$ für die initiale Registerbelegung c$[n_1, ..., n_k]$ mit der Registerbelegung c terminiert.

Dabei ist k die Anzahl an Eingaberegistern und Register i das Ausgaberegister von $\mathcal{R}$.

1.4 RM-berechenbar

Eine partielle Funktion

$$f_{\mathcal{R}} : \mathbb{N}^k \to \mathbb{N}$$

wird RM-berechenbar genannt, wenn es eine Registermaschine $\mathcal{R}$ gibt, so dass $f = f_{\mathcal{R}}$. Alle partiellen Funktionen, für die man ein Programm in irgendeiner Programmiersprache schreiben kann, sind RM-berechenbar, da diese Programmiersprache mit einem Compiler in ein Registermaschinenprogramm übersetzt wird.

1.5 Beispiele zur Benutzung von RM's:

1.5.1 Graphendarstellung

Ein endlicher Graph wird wie folgt dargestellt:

- n=|v|= Knotenanzahl wird in Register 1 gespeichert.
- Die Adjazenzmatrix wird so gespeichert: Register $(i-1)*n+j+1$ enthält den Eintrag der i-ten Zeile und j-ten Spalte.
  Stellt man sich eine Adjazenzmatrix vor, so werden die Zeilen dieser getrennt und hintereinander gereiht.

1.5.2 Lineare Listen

- In Register R1 wird die Position des Listenkopfs gespeichert
- R2 und R3 werden für Nebenrechnungen freigehalten *(Keine Ahnung für welche)*
- gerade Register R4,R6,... werden für die Listenelemente benutzt
- ungerade Register R5,R7,... werden für die Verkettung benutzt

(In Register $2i$ steht der Wert des Listenelements. In Register $2i+1$ steht die Position des nächsten Listenglieds.)
Die Darstellung ist ähnlich der der Darstellung von Listen mit Arrays. Es wird immer ein Listenelement gespeichert und an zweiter Stelle die Position des folgenden Elementes.

1.6 Kostenmaße für RM's:

1.6.1 Allgemeines

Die wesentlichen Faktoren, an denen die Effizienz von Algorithmen gemessen wird, sind die Laufzeit und der benötigte Speicherplatz. Für RM kann diese durch die Funktionen

$t_{\mathcal{R}}$	logarithmische Zeit
$t^u_{\mathcal{R}}$	uniforme Zeit
$s_{\mathcal{R}}$	logarithmischer Platz
$s^u_{\mathcal{R}}$	uniformer Platz

beschrieben werden. (Dabei steht t steht für time, s für space und u für uniform.)
Bei der **Zeitkomplexität** werden die abgearbeiteten **Befehle** gezählt, bei der **Platzkomplexität** die besuchten **Registerzellen**.

1.6.2 Uniformes Kostenmaß

Das uniforme Kostenmaß benutzt Einheitskosten für die RM-Befehle und die Registerzellen.

- $t^u_{\mathcal{R}}$ ($n_1, ..., n_k$) ist die Anzahl an RM-Befehlen die $\mathcal{R}$ für das Eingabetupel $(n_1, ..., n_k)$ ausführt, bis der Befehlszähler den Wert b = ∞ erreicht. Ein Befehl der mehrfach abgearbeitet wird, zählt auch mehrfach.
- $u^u_{\mathcal{R}}$ ($n_1, ..., n_k$) ist die Anzahl an Registern, auf die $\mathcal{R}$ für das Eingabetupel $(n_1, ..., n_k)$ zugreift, bis der Befehlszähler den Wert b = ∞ erreicht. Jede benutzte Registerzelle zählt einfach.

Das uniforme Kostenmaß ist nur realistisch, wenn die Darstellungsgröße der Daten nicht stark anwächst. (Zum Beispiel realistisch beim Sortieren, aber nicht beim Finden von Primzahlen)

1.6.3 Logarithmisches Kostenmaß

Es wird die Darstellungsgröße der in den Registern gespeicherten Werte berücksichtigt.
Logarithmische Länge: Die Logarithmische Länge $L(n)$ ist die Anzahl an Bits, die benötigt werden, um n als Binärzahl darzustellen. Die Anzahl der Bits, die für eine Zahl n benötigt werden errechnet sich wie folgt:

$$L(n) = \begin{cases} 1 & falls\, n = 0 \\ \lfloor log_2(n) \rfloor + 1 & falls\, n \leq 0 \end{cases}$$

Zeitbedarf: Die Kosten werden durch aufsummieren **aller** zur Ausführung des betreffenden Befehls benötigten Größen ermittelt. Dabei werden Marken und Sprungbefehle mit 1 bewertet.
Speicherplatzbedarf: Die Speicherkosten jeder benutzten Registerzelle werden aufsummiert. Anders als beim Zeitbedarf wird die Registerzelle nicht erneut aufsummiert, wenn die Zelle mehrmals besucht wird.
Kosten in Abhängigkeit der Eingabegröße: Die Eingabegröße ist die Summe aller logarithmierten Eingabewerte. (z.B. wenn man nur einen Wert $n \leq 1$ hat, ist die logarithmierte Summe $log_2(n)$, da nur ein Wert vorhanden ist. Kostenfunktionen in abhängig der Eingabegröße werden in Großbuchstaben angegeben. (T und S statt t und s)

1.6.4 Beispiel der Anwendung des logarithmischen Kostenmaßes

Befehl	logarithmische Zeit
LOAD i	$L(i) + L(c(i))$
STORE $*i$	$L(c(0)) + L(i) + L(c(i))$
ADD i	$L(i) + L(c(0)) + L(c(i))$
SUB #4	$L(c(0)) + L(4) = L(c(0)) + [2 + 1]$
GOTO b	1
IF $c(0) > 9$ **THEN** GOTO b **FI**	$L(c(0)) + L(9) + 1 = L(c(0)) + [3 + 1] + 1$
END	1

1.6.5 Polynomielle Beschränktheit

Die Schreibweise

$$T(n) = O(poly(n))$$

bedeutet, daß es ein Polynom p gibt mit

$$T(N) \leq p(n)$$

für alle $n \in \mathbb{N}$ Man beachte, daß die Beschränkung durch ein Polynom nicht impliziert, daß auch die entsprechende Fkt. ein Polynom ist.
In analoger Weise ist die exp. Beschränktheit $O(2^{poly(n)})$ definiert.

1.6.6 Polynomielle Zeitbeschränktheit

Wir nennen $\mathcal{R}$ polynomiell zeitbeschränkt, falls gilt

$$T_{\mathcal{R}}(N) = O(poly(N)).$$

Analoge Bedeutung haben die Begriffe **polynomielle Platzbeschränkung** und **exponentielle Zeitbeschränkung**.

1.6.7 Effizienz von Algorithmen

Als effizient gelten Algorithmen, deren Laufzeit unter dem logarithmischen Kostenmaß durch ein Polynom beschränkt ist.

2 Turingmaschinen

2.1 DTM - Deterministische Turingmaschine

2.1.1 Definition

Eine DTM ist ein 7er-Tupel $\tau = (Q, \sum, \Gamma, \delta, q_0, \square, F)$ wobei

- Q Zustandsmenge (endlich)
- $\sum$ Eingabealphabet
- Γ Bandalphabet, wobei $\Gamma \supseteq \sum$
- q_0 Startzustand $q_0 \in Q$
- F Endzustandsmenge $F \subseteq Q$
- $\square$ Blanksymbol $\square \in \Gamma$
- δ partielle Übergangfkt.

Die partielle Übergangsfunktion wird aus mehreren kleinen Übergangsfunktionen zusammengesetzt von denen eine wie folgt beschrieben ist:

$$\delta(q, a) = (p, b, move)$$

Wobei q der Startzustand dieser Übergansfunktion ($q \in Q$), a das Startzeichen ($a \in \Gamma$), p der Zustand, in den diese Übergangsfunktion überführt ($p \in Q$), b das Zeichen, in den diese Übergangsfunktion überführt ($b \in \Gamma$) und $move \in \{L, N, R\}$ als Bewegung des Zeigers, wobei L für Bewegung nach links, N für keine Bewegung, R für eine Bewegung nach rechts steht.

2.1.2 Verhalten

Die DTM **akzeptiert**, wenn sie in einen Endzustand gerät, egal, ob die Übergangsfunktion noch weitere Zustandswechsel erzeugen kann oder nicht. Das Ausgabewort steht dann rechts vom Lese/Schreibkopf. Sie **verwirft**, wenn der aktuelle Zustand kein Endzustand ist und die partielle Übergangsfunktion an dieser Stelle nicht definiert ist und somit keine weiteren Zustandswechsel erzeugen kann.
$\delta(q, a) = \perp$ steht nur zur einfacheren Lesbarkeit in manchen DTMs, daß die Übergangsfunktion an dieser Stelle nicht definiert ist und hier ein Abbruch beabsichtigt ist. Wenn q hierbei wieder Endzustand ist, akzeptiert die DTM, anderfalls nicht.
Es gibt auch sogenannte **Fangzustände**. In solchen Zuständen läuft die DTM endlos auf einer Stelle ($\delta(q, a) = (q, a, N)$). Formal kann man solche Zustände durch entsprechende verwerfende Zustände ersetzen. Wenn die DTM also in einen Fangzustand kommt, verwirft sie sozusagen auch.

2.1.3 Kostenmaß für Turingmaschinen

Zeit:
$t_\tau(w)$ = Anzahl der Konfigurationswechsel, die τ bei der Eingabe w durchführt.
Platz:
$s_\tau(w)$ = Anzahl der Bandzellen, die τ bei der Eingabe w besucht.

2.1.4 Konfigurationsrelation

Die Konfigurationsrelation dient dazu, aufzuschreiben, was die DTM „macht". Sie wird folgendermaßen geschrieben:

$$Startband \vdash Band_1 \vdash Band_2 \vdash ... \vdash Band_k \vdash^* Band_l ... \vdash \perp$$

Steht das $\vdash$-Zeichen ohne Sternchen oben, so darf die DTM eine Übergangsfunktion abarbeiten. Steht ein Sternchen oben, so ist eine Kette von Abarbeitungen der Übergangsfunktion gemeint.
$Band_x$ sieht dabei folgendermaßen aus: Zuerst kommt das Band, welches vor dem aktuellen Zeiger steht. Dann kommt der aktuelle Zustand. Dannach die Bandzelle auf die der aktuelle Zeiger zeigt. Dannach der Rest des Bandes. Beispiel:

$$\#010q_51\#$$

Der Zeiger steht hierbei auf der letzten 1.

2.1.5 Linksseitig beschränktes Band

Wir können DTMs beschränken, indem wir das Band an einer Seite abschneiden, so daß nur eine Seite ins Unendliche geht. Wir können jedoch dieses Band wieder zu einem unendlichen Band machen, indem wir zwischen jeder Bandzelle eine Bandzelle aus der linken Seite, die nicht mehr unendlich ist hineinfügen. Dies funktioniert so ähnlich wie ein Reißverschluß. Beispielsweise sieht das Band wie folgt aus:

$$\leftarrow l_4 l_3 l_2 l_1 r_1 r_2 r_3 r_4 \rightarrow$$

Wenn wir es zwischen l_1 und r_1 schneiden, sieht es so aus:

$$|r_1 l_1 r_2 l_2 r_3 l_3 r_4 l_4 \rightarrow$$

2.1.6 Hintereinanderschaltung

Wir können Turingmaschinen hintereinander schalten, sowie sie auch mit Schleifen und bedingten Anweisungen usw. verknüpfen.

2.1.7 Simulation von Registermaschinen mit Turingmaschinen

Wir können Registermaschinen mit Turingmaschinen simulieren:
Zu jeder Registermaschine R gibt es eine Turingmaschine τ, so daß die durch R berechnete partielle Funktion $f_R : \mathbb{N}^k \to \mathbb{N}$ mit der durch τ berechneten partiellen Fkt. übereinstimmt $f_\tau : \mathbb{N}^k \to \mathbb{N}$.

Die Simulation funktioniert wie folgt: Für jeden Befehl der im Programm der Registermaschine steht, gibt es eine **Zustandsmenge**. Der Befehlszähler wird also dadurch ersetzt, daß der aktuelle Zustand in einer der Zustandsmengen ist. Dann benötigt die Turingmaschine zum simulieren vier Bänder (Erklärung Mehrband-Turingmaschinen weiter unten) Auf dem ersten Band befindet sich die **Eingabe**, auf dem zweiten Band die **Register und Akkumulatorbelegung**, auf dem dritten die **Ausgabe** und das vierte Band ist für **Nebenrechnungen** reserviert.
Wird nun ein Befehl ausgeführt, wie zum Bespiel *add 5*, so wechselt die Turingmaschine zu Beginn in die Zustandsmenge für diesen Befehl. Nun kann es los gehen. Aus dem Register 5, welches auf Band 2 gespeichert ist, wird die Zahl ausgelesen und auf das Band für Nebenrechnungen kopiert. Dann wird zum Akkumulator gespult und dieser mit binärer Addition mit dem Nebenrechneband addiert und der Wert zurück in den Akkumulator geschrieben.

Für **rekursive Funktionsaufrufe** müssen wir ein **weiteres Extraband** reservieren, um auf diesem die Rekursion zu verfolgen.

Die **Zeitkomplexität** der Simulation ist **polynomiell**. Dies kommt daher zustande, daß alle Registermaschinenbefehle in polynomieller Laufzeit abgearbeitet werden können. Das Spulen auf den Bändern geschieht auch in polynomieller Laufzeit.

2.1.8 Simulation von Turingmaschinen mit Registermaschinen

Ebenso können wir Turingmaschinen mit Registermaschinen simulieren. Dies können wir auf zwei verschiedene Arten tun:

- Mit indirekter Adressierung: Wir bilden das Band auf die Register ab. Wir haben gezeigt, daß eine linkseitig beschränkte Turingmaschine gleich einer unbeschränkten Turingmaschine ist. Wir lassen das Band an einer Stelle in den Registern starten. In speziell reservierten Registern vor dieser Stelle speichern wir den aktuellen Zustand, die aktuelle „Band-Register-Zelle" usw.

- Mit zwei Kellern: Der eine Keller beinhaltet den linken Teil des Bandes, der andere Keller beinhaltet den rechten Teil des Bandes. Je nachdem nehmen wir eine Bandzelle von dem einen Kellerspeicher und tuen die aktuelle auf den anderen oder umgekehrt. *Über den Umweg über die Turingmaschine und wieder zurück zur Registermaschine läßt sich beweisen, daß eine Registermaschine ohne indirekte Adressierung immernoch die gleiche Mächtigkeit hat, wie eine Registermaschine mit indirekter Adressierung.*

Wir realisieren jeweils den Zustandwechsel und das Schreiben des neuen Zeichens mit nacheinanderfolgenden IF-Abfragen.
(z.B.: IF (Zustand=1 und Zeichen=0) THEN Zustand = 2; Zeichen =1; einenNach-LinksBewegen FI)

2.1.9 Churchsche These

Church hat die These aufgestellt, daß die durch Turingmaschinen bzw. Registermaschinen berechnenbaren partiellen Funktion genau mit der Klasse der intuitiv berechenbaren Funktionen übereinstimmt. Dadurch erhalten wir durch solche Maschinen ein gutes Bild, was auch zukünftige Rechner, die vielleicht mit anderen Modellen rechnen, leisten werden.

2.2 Mehrbandige Turingmaschine

2.2.1 Definition

Die Definition ähnelt der, der einbandigen DTM. Hinzukommt, daß die Übergangsfunktion δ anders aussieht:

$$\delta : Q \times \Gamma^k \rightarrow Q \times (\Gamma\{L, N, R\})^k$$

Somit sieht ein Teil der Übergangsfunktion folgendermaßen aus:

$$\delta(q, (a_1, a_2, ..., a_k)) = (p, < b_1, move_1 >, < b_2, move_2 >, ..., < b_k, move_k >)$$

Die Konfiguration wird in Spaltenvektorschreibweise geschrieben.

2.2.2 Überführung in eine Einbandturingmaschine

Jede mehrbandige Turingmaschine kann in eine einbandige Turingmaschine überführt werden. Dies wird dadurch erreicht, daß die Einbandtouringmaschine neue Zeichen im Bandalphabet bekommt und zwar so, daß alle Konstellationen der vorkommenden Zeichen der Mehrbandturingmaschine ein Zeichen auf der Einbandturingmaschine bekommen, so daß alle Bänder zu einem Band zusammengeschmolzen werden können.
Die Kosten der Simulation einer Mehrbandturingmaschine auf einer Einbandturingmaschine sind $C \cdot N_w \cdot Mw$, wobei N_w die gesamte Anzahl an Bandquadraten, die bei der Berechnung der Mehrbandturingmaschine benutzt werden, M_w die Anzahl der Konfigurationswechsel der Mehrbandturingmaschine und C eine Konstante abhängig von w.

2.3 NTM - Nichtdeterministische Turingmaschine

2.3.1 Definition

Die nichtdeterministische Turingmaschine wird so definiert, wie die deterministische. Nur gibt es in ihr Konfigurationen zu der mehrere Nachfolgekonfigurationen existieren. Die NTM wählt aus den Nachfolgekonfigurationen intuitiv dann die Richtige aus. Man kann sich anschaulich dies als eine Art „Orakel" vorstellen, welches ohne Berechnung die richtige in diesem moment passende Lösung liefert. Eine Übergangsfunktion sieht wie folgt aus: (k: Anzahl der möglichen Berechnungen)

$$\delta(q, b) = \{(q_1, b_1, move_1), ..., (q_k, b_k, move_k)\}$$

2.3.2 Konfigurationsrelation

Anders als bei der Konfigurationsrelation der DTMs gibt es hier mehrere mögliche Berechnungen. Wir zeichen so einen Konfigurationsbaum. Dabei müssen wir beim Zeichnen darauf achtgeben, daß wir bei unendlichen sich wiederholende Berechnungen Punkte, die andeuten, daß dies eine unendliche Berechnung ist, unter der jeweiligen Konfigurationswechsel machen. Wir dürfen keine Schleifen im Baum kreiren. Gibt es eine Blatt bei der die NTM akzeptiert, so akzeptiert die gesamte NTM für dieses Eingabewort. Gibt es kein Blatt, für welches die NTM akzeptiert, so verwirft sie.

2.3.3 Simulation mit einer DTM

Zu beachten ist, daß es in der Natur keine NTMs gibt. Man hat noch keinen Rechner gebaut, der so etwas machen kann (Es wäre schön, wenn es einen solchen gäbe, da er NP-Probleme in polynomieller Zeit lösen könnte). Aber wir können NTMs auf DTMs simulieren. Dazu lassen wir anschaulich durch den Konfigurationsrelationsbaum eine Breitensuche (zur Erinnerung: Breitensuche läuft mit einer Queue) laufen. Wenn wir ein akzeptierendes Blatt finden, dann akzeptiert die NTM. Wenn die NTM nicht terminiert oder wenn sie terminiert und alle Blätter verwerfen, verwerfen wir.

3 Entscheidungprobleme

Viele Algorithmen beinhalten ein Entscheidungsproblem. So muß man zum Beispiel entscheiden, ob eine Zahl x eine Primzahl ist.

3.1 Definitionen: Entscheidbarkeit

Gegeben ist
eine Sprache $L \subseteq \Sigma^*$
und ein Wort $w \in \Sigma^*$.
Gefragt ist ob $w \in L$?

3.1.1 Entscheidbar

Sei $L \subseteq \Sigma^*$ eine Sprache. L heißt *entscheidbar* oder *rekursiv* wenn es eine DTM τ mit

$$L(\tau) = L$$

gibt, die

$$w \in \overline{L(\tau)}$$

verwirft. Alle Wörter $w \notin L$ werden von der DTM verworfen.
In Worten: Alle Wörter $w \in L$ werden von der DTM mit „JA" akzeptiert. Alle Wörter $w \notin L$ müssen von der DTM mit „NEIN" verworfen werden.

3.1.2 Semientscheidbar

Sei $L \subseteq \Sigma^*$ eine Sprache. L heißt *semientscheidbar* wenn es eine DTM τ gibt, für die gilt:

$$L(\tau) = L$$

Für alle $w \in \overline{L(\tau)}$ wird keine Aussage gemacht. Die DTM kann entweder verwerfen oder endlos laufen.
In Worten: Alle Wörter, die durch die Sprache L erzeugt werden können, müssen von

der DTM mit „JA" akzeptiert werden. Es wird keine Aussage darüber gemacht, ob die DTM verwirft oder endlos läuft, wenn ein Wort w eingebeben wird, für das gilt: $w \in \Sigma^* \backslash L$.
Somit ist eine entscheidbare Sprache gleichzeitig auch immer semientscheidbar, da sie für alle Worte $w \in \Sigma^* \backslash L$ immer verwirft.

3.1.3 Unentscheidbar

Eine Sprache $L \subseteq \Sigma^*$ ist *unentscheidbar*, wenn es keine DTM τ gibt, für die gilt:

$$L(\tau) = L$$

3.1.4 Zusätze

(i) L entscheidbar $\Leftrightarrow$ $\overline{L}$ entscheidbar
(ii) Wenn die Sprachen $L \subseteq \Sigma^*$ und $\overline{L} = \Sigma^* \backslash L$ semientscheidbar sind, dann ist L entscheidbar. Wir können die DTMs für die beiden Sprachen parallel schalten. Eine davon wird für ein Wort w akzeptieren. Akzeptiert die DTM für L so ist $w \in L$. Akzeptiert die DTM für $\overline{L}$ so ist $w \notin L$.
(iii) L semientscheidbar $\Leftrightarrow$ L rekursiv aufzählbar (Definition rekursive Aufzählbarkeit weiter unten)
„$\Leftarrow$": Ob ein Wort $w \in L$ ist, testet man, indem man alle Worte von L rekursiv aufzählt. Errechnen wir w, so ist $w \in L$. Das Semientscheidungsverfahren gibt in diesem Fall „JA" zurück. Wenn das Wort w nie berechnet wird, laufen wir endlos. Deshalb ist L nur semientscheidbar und nicht entscheidbar.
„$\Rightarrow$": Wir bedienen uns hierbei eines Tricks. Wir legen eine obere Schranke fest, ab der wir sagen, daß $w \notin L$. Wir können nun alle Wörter mit dem ersten Wort beginnend, ob sie Element von L sind. Diese Wörter können wir dann aufzählen.

3.2 Rekursive Aufzählbarkeit

Eine Sprache L ist rekursiv aufzählbar, wenn $L = \{\}$ oder wenn es eine totale Funktion $f : \mathbf{N} \to \Sigma^*$, so daß

$$L = \{f(0), f(1), f(2), ..., f(n)\} = f(\mathbb{N})$$

In Worten: Es gibt ein Verfahren, welches die Sprache Wort für Wort aufzählt. Zum Beispiel ist die Sprache der Primzahlen aufzählbar, weil es ein Verfahren gibt, welches die Primzahlen nacheinander aufzählt. Auch wenn dieses nicht enden wird, wird es trotzdem jede Primzahl p nach einer gewissen Zeit t_p errechnen.

3.3 Halteproblem

3.3.1 Satz: Unentscheidbarkeit des Halteproblems:

Die Sprache des speziellen Halteproblems

$$H' = \{w\#x \in \{0,1\}^*, \tau_w \textit{hält bei der Eingabe x an}\}$$

ist unentscheidbar.

3.3.2 Intiutives Verfahren

Das intiutive Verfahren läuft über eine **Halteproblem-Tabelle**. In dieser bekommt jder Algorithmus eine Zeile und jedes Eingabewort eine Spalte. In jeder Zelle steht, ob der Algorithmus für die Eingabe terminiert oder nicht. Terminiert er, finden wir hier ein „J" andernfalls ein „N".
Der **Halteproblemalgorithmus** berechnet die Halteproblemtabelle. Er soll **terminieren**, wenn er ein **„N"** in der Halteproblemtabelle berechnet und **endlos laufen**, wenn er ein **„J"** berechnet. Halteproblemalgorithmus und der eigentliche Algorithmus werden parallel gestartet. Terminiert der Halteproblemalgorithmus zuerst, so terminiert der Algorithmus nicht. Anderfalls wird dem Algorithmus unendlich viel Zeit gelassen, um das Problem zu lösen und dann zu terminieren, da der Halteproblemalgorithmus unendlich läuft.
Betrachten wir nun das **Komplementär der Diagonalen** der Halteproblemtabelle. (Komplement ist immer genau der Umgekehrte Wert, der in der Tabelle steht.) Wir betrachten auf der Diagonalen immer das Eingabewort i und den Algorithmus A_i
Da der **Halteproblemalgorithmus auch ein Algorithmus** ist und **in der Halteproblemtabelle** vorkommt, ist der mit der Eingabe i **zu testende Algorithmus irgendwann selbst der Halteproblemalgorithmus**.

- Nach der Konstruktion des Algorithmusses steht hier ein „J", wenn er terminiert, ein „N", wenn er nicht terminiert.
- Das Komplement für den Halteproblemalgorithmus, dieses Algorithmusses ist aber ein „N", wenn der Algorithmus terminiert und ein „J", wenn der Algorithmus endlos läuft, so daß er abgebrochen werden kann.

Dies ist ein Widerspruch, da ein und der selbe Algorithmus für die selbe Eingabe i **nicht endlos laufen kann und gleichzeitig terminieren kann**. Der Halteproblemalgorithmus kann unmöglich selbst entscheiden, ob er terminiert. Damit gibt es keinen solchen Algorithmus.

3.3.3 Universelle Turingmaschinen

Mit Hilfe einer universellen Turingmaschine kann man eine Turingmaschine simulieren. Dies erfolgt folgendermaßen, indem man ein Eingabewort in diese Turingmaschine eingibt, welches wie folgt aussieht:
#bin(Anzahl der Zustände)#bin(Anzahl der Bandsymbole)#Codes der Übergangsfunktionen##Codes der Übergangsfunktionen, die in einem Endzustand enden#
Dabei kann eine Übergangsfunktion $\delta(q_i, a_j) = (q_I, a_J, X)$ folgendermaßen dargestellt werden:

$$\#\#bin(i)\#bin(j)\#bin(I)\#bin(J)\#bin(X)\#\#$$

Dabei ist $bin(L) = 00$, $bin(R) = 01$ und $bin(N) = 10$

3.3.4 Semientscheidbarkeit des Halteproblems

Die Sprache des Halteproblems ist

$$H = \{w\#x : w, x \in \{0,1\}^*, \tau_w \, hält \, bei \, Eingabe \, x \, an\}$$

Wir konstruieren eine Turingmaschine τ, die die Turingmaschine τ_w simuliert. Zunächst wird überprüft, ob das Eingabewort für die Turingmaschine τ die Gestalt $w\#x$ hat. Hat

es diese nicht, so verwirft τ. Dann wird die Turingmaschine τ_w simuliert. Akzeptiert sie, dann kann auch τ akzeptieren. Azeptiert sie nicht und läuft endlos so läuft auch τ endlos.
$\Rightarrow$ Die Sprache H des Halteproblems ist semientscheidbar.

3.3.5 Spezielles Halteproblem

Die Sprache des speziellen Halteproblems ist

$$H' = \{w \in \{0,1\}^* : \tau_w \, hält \, bei \, Eingabe \, w \, an\}$$

Diese Sprache ist unentscheidbar.
Wir sehen hier deutlich, daß $\tau_w = \tau$. Das heißt, daß das spezielle Halteproblem selbst entscheiden soll, ob es anhält oder nicht. Da die Haltemaschine selbst ein Algortihmus ist, muß es möglich sein, diese Selbstanwendung durchzuführen.

τ' entscheidet die Sprache. Ist ein Wort $w \in H'$ so akzeptiert τ' andernfalls verwirft τ'. Wir simulieren τ' mit einer Turingmaschine τ.

- Akzeptiert τ' das Wort w, geht τ in einen Fangzustand. (Laufe endlos und verwerfe somit)
- Läuft τ' endlos, d.h. es verwirft, dann hält τ akzeptierend an.

	τ hält bei Eingabe w an
$\Leftrightarrow$	τ' die Eingabe w verwirft
$\Leftrightarrow$	$w \notin H'$
$\Leftrightarrow$	τ_w hält bei der Eingabe w nicht an

Die erste Zeile ergibt zu der letzten einen Widerspruch, da $\tau_w = \tau$.

Daraus folgt, daß die Sprache des speziellen Halteproblems unentscheidbar ist.

Wir können diesen Beweis auch umgekehrt vormulieren:

	τ läuft bei Eingabe w endlos
$\Leftrightarrow$	τ' die Eingabe w akzeptiert
$\Leftrightarrow$	$w \in H'$
$\Leftrightarrow$	τ_w hält bei der Eingabe w an

Da wieder $\tau = \tau_w$ ist dies ein Wiederspruch und somit können wir wiederum sagen: Die Sprache des speziellen Halteproblems ist unentscheidbar.

3.3.6 Weiteres

Daß das allgemeine Halteproblem unentscheidbar ist, können wir zeigen, indem wir es reduzieren: $H \leq H'$. Wir finden eine Funktion $f(x) = x\#x$, die das allgemeine Halteproblem auf das spezielle reduziert.
Das Komplement $\overline{H}$ der Sprache des Halteproblems ist nicht semientscheidbar.

3.4 Satz von Rice

Definition: Sei $\mathcal{S}$ eine nichtleere, echte Teilmenge der Menge aller partiellen berechenbaren Funktionen $f : \{0,1\}^* \to \{0,1\}^*$. Dann ist die Sprache

$$L_{\mathcal{S}} = \{w \in \{0,1\}^* : \tau_w \text{ berechnet eine partielle Funktion } f \in \mathcal{S}\ \}$$

unentscheidbar.
[... d.h., die Sprache, die alle Turingmaschinenprogramme enthält, die eine Funktion aus $\mathcal{S}$ berechnen, ist unentscheidbar.]

Beweis: durch Fallunterscheidung: 1. $f_\perp \in \mathcal{S}$ oder 2. $f_\perp \notin \mathcal{S}$
Beweis 1. Fall:
$f_\perp :=$ überall undef. part. Funktion $\{0,1\}^* \to \{0,1\}^*$.
[also ist die überall undef. Funktion einmal in der Teilmenge $\mathcal{S}$ und einmal nicht.]
z.z.: $\overline{H_\epsilon} \leq L_{\mathcal{S}}$ und $H \leq H_\epsilon$

$$H_\epsilon = \{w \in \{0,1\}^* : \tau_w \text{ hält bei Eingabe } \epsilon \text{ an}\}$$

$$\overline{H_\epsilon} = \{w \in \{0,1\}^* : \tau_w \text{ hält bei Eingabe } \epsilon \text{ nicht an}\}$$

Zunächst wählen wir uns eine part. berechenbare Fkt. aus, die nicht in $\mathcal{S}$ liegt, also so dass gilt:

$$f : \{0,1\}^* \to \{0,1\}^* \text{ mit } f \notin \mathcal{S}.$$

Sei τ eine DTM, die diese Funktion berechnet.
Nun definieren wir uns eine neue DTM τ_w', welcher wir jedes neue Wort $w \in \{0,1\}^*$ zuordnen, so dass durch $w \mapsto code(\tau_w')$ eine tot. ber. Funktion mit

$$w \in \overline{H_\epsilon} \text{ gdw } code(\tau_w')$$

gegeben ist. Die DTM τ_w' sei für eine Eingabe $x \in \{0,1\}^*$ wie folgt definiert: τ_w' berechnet die Fkt.

$$f_w = \begin{cases} f_\perp & : \quad falls\, \tau_w\, bei\, leerem\, band\, nicht\, anhält \quad [simuliert\, also\, \overline{H_\epsilon}] \\ f & : \quad sonst. \end{cases}$$

[Nun wird auch klar, warum wir oben ein DTM τ definiert haben, die eine Funktion f berechnet. Da nach Ann. die überall undefinierte Funktion in der Teilmenge $\mathcal{S}$ liegt, brauchen wir eine andere Funktion, die eine endlose bzw. verwerfende Berechnung simuliert. Dies können wir ohne weiteres mit der Funktion f tun, da wir diese so designed haben, dass sie nicht in $\mathcal{S}$ liegt.]

Da wir nun wissen, dass $f_\perp \in \mathcal{S}$ und $f \notin \mathcal{S}$ gilt können wir daraus folgern:

$w \in \overline{H_\epsilon}$		
gdw	τ_w hält bei leerem Band nicht an	[def. $\overline{H_\epsilon}$]
gdw	τ_w' berechnet $f_w = f_\perp$	[def. τ_w']
gdw	τ_w' berechnet partielle Fkt. in $\mathcal{S}$	[da $f_\perp \in \mathcal{S}$ n.Ann.]
gdw	$code(\tau_w') \in L_{\mathcal{S}}$	[def. τ_w']

$\Rightarrow \overline{H_\epsilon} \leq L_{\mathcal{S}}$
[Uns ist also eine Reduktion geglückt, eine Eingabe $w \in \{0,1\}^*$ für $\overline{H_\epsilon}$ in ein Eingabecode für (τ_w') für $L_{\mathcal{S}}$ umzuwandeln.]

Nun muß noch gezeigt werden, dass $\overline{H_\epsilon}$ unentscheidbar ist. Daraus würde dann die Unentscheidbarkeit für $L_\mathcal{S}$ folgen.
[Wenn man zeigen will, dass das Komplement einer Sprache unentscheidbar ist, reicht es dies für die Sprache (in diesem Fall H_ϵ) selber zu zeigen. (vgl. Satz 2.1.5) Wir würden somit erhalten: $H \leq H_\epsilon$ Wir wissen von oben: $\overline{H_\epsilon} \leq L_\mathcal{S} \Rightarrow H \leq L_\mathcal{S}$. Die Sprache H des Halteproblems ist bekannterweise unentscheidbar!]
Sei $y \in \{0,1\}^*$. Wir definieren uns eine DTM τ_y'', deren Arbeitsweise wie folgt festgelegt ist:

- τ_y'' gestartet auf ein nichtleeres Eingabewort hält verwerfend an
- τ_y'' gestartet auf dem Eingabewort ϵ verhält sich so:
  - Ist y nicht in der Form $y = w\#x$, wobei $w, x \in \{0,1\}^*$, dann läuft τ_y'' endlos.
  - Ist $y = w\#x$, wobei $w, x \in \{0,1\}^*$, dann simuliert τ_y'' die DTM τ_w bei Eingabe x. In diesem Fall hält τ_y'' bei Eingabe ϵ genau dann an, wenn τ_w bei Eingabe x anhält.

[Mit anderen Worten: die DTM τ_y'' ist so gebaut, dass alle Eingaben (die das Halteproblem ja als Eingabe Akzeptieren muß) die nicht von der DTM τ_w(die die Sprache H_ϵ entscheidet) akzeptiert werden, verwirft. Dies muß so sein, damit die Reduktion funktioniert.]

Wir erhalten somit:

$w\#x \in H$	
gdw	τ_w hält bei Eingabe x an [def. τ_w]
gdw	$\tau''_{w\#x}$ hält für die Eingabe ϵ an. [def. τ_y'']
gdw	$code(\tau''_{w\#x}) \in H_\epsilon$[nach def. von H_ϵ]

Die Reduktion $H \leq H_\epsilon$ ist somit ebenfalls geglückt.
Resultat: Die Sprache $L_\mathcal{S}$ ist unentscheidbar !! der 2.Fall der Fallunterscheidung klappt ähnlich, indem man zeig: $H_\epsilon \leq L_\mathcal{S}$.

4 Komplexität

4.1 Drei Varianten eines Problems haben aus komplexitätstheoretischer Sicht denselben Schwierigkeitsgrad

Man kann Probleme wie „TSP", „RP" oder „GFP" in drei Varianten formulieren:

1. Entscheidungsproblem: Gibt es eine Lösung mit dem Wert k (Als Beispiel TSP: Können alle Städte mit Kosten k besucht werden)
2. Optimierungsproblem (Typ 1): Berechne den Wert der optimalen Lösung (Als Beispiel TSP: Gibt den Wert k der Kosten an, die für eine kürzeste Rundreise benötigt werden)
3. Optimierungsproblem (Typ 2): Bestimme die optimale Lösung (Als Beispiel TSP: Gib die Knoten der kürzesten Rundreise in der Reihenfolge zurück)

4.2 DTIME und NTIME

Es gibt zwei verschiedene Zeitangaben. Die eine ist die deterministische Zeit (DTIME), die andere ist die nichtdeterministische Zeit (NTIME). DTIME und NTIME für einen Algorithmus sind verschieden, da bei der Benutzung von NTIME die Berechnung abgekürzt wird, indem geraten wird.
Als Beispiel nehme ich den Primzahlalgorithmus. Wir überprüfen, ob eine Zahl n eine Primzahl ist. Dieser hat in NTIME die Laufzeit (unter dem logarithmischen Kostenmaß)

$$\Theta(L(n)) = \Theta(log(n))$$

Wir raten mit Hilfe eines Orakels eine Zahl k für die gilt $1 < k < n$. Ist n keine Primzahl, so ist n durch k ganzzahlig teilbar, da wir genau einen Teiler erraten haben. Ist n eine Primzahl, so ist n nicht durch die nichtdertiministisch bestimmte Zahl k teilbar. Dies geht in $\Theta(L(n)) = \Theta(log(n))$, da das ganzzahlige Teilen in $\Theta(L(n) + L(m)) = \Theta(L(n))$ läuft.

DTIME für den Primzahltest ist jedoch

$$\Theta(2^{L(n)}L(n)) = \Theta(n \log n)$$

Dies ist deshalb der Fall, weil wir n-Mal teilen müssen, um festzustellen, ob n durch irgendeinen Teiler $k < n$ teilbar ist.

4.3 P, NP und EXPTIME

4.3.1 Definition

Wir definieren die Klassen P, NP und EXPTIME wie folgt:

$$P = PTIME = \bigcup_{k \geq 1} DTIME(n^k)$$

Das heißt, daß in PTIME alle Klassen liegen, die in deterministischer Zeit mit einem Polynom darstellbar sind. Das k ist hierbei fest zu wählen. Beispielsweise sind $\Theta(1)$,$\Theta(n)$,$\Theta(log\, n)$,$\Theta(n^2)$ oder $\Theta(n^{100000}) \in PTIME$.

$$NP = NPTIME = \bigcup_{k \geq 1} NTIME(n^k)$$

Das heißt, daß alle Probleme, die sich in polynomieller Zeit mit einem nichtdeterministischen Algorithmus lösen lassen, in NP sind.

$$EXPTIME = \bigcup_{c>1} \bigcup_{k \geq 1} DTIME(c^{n^k})$$

In EXPTIME ist das c und das k fest, wobei n läuft. Element von EXPTIME ist z.B. $\Theta(2^{n^1}) = \Theta(2^n)$
Analog läßt sich auch NEXPTIME definieren:

$$NEXPTIME = \bigcup_{c>1} \bigcup_{k \geq 1} NTIME(c^{n^k})$$

4.3.2 NP läßt sich in exponentieller PTIME lösen

Wir nehmen eine DTM, die die NTM simuliert. Dabei schauen wir uns den Berechnungsbaum an. Dieser hat höchsten den Verzweigungsgrad c. Dieses c ist beschränkt, da es endlich viele mögliche Berechnungen pro partieller Funktion gibt: $c < |Q| * |\Gamma| * 3$ (3 wegen $\{L, N, R\}$) Somit gilt:

$$O(c^{Höhe\,des\,Baumes})$$

Die Höhe des Baumes ist hierbei variabel und von Eingabewort zu Eingabewort unterschiedlich. Das c bleibt jedoch fest. Deshalb kann der ganze Baum in exponentieller Zeit besucht werden.

4.3.3 Polynomialzeit-Akzeptanz einer NTM

Wenn der Berechnungsbaum für ein akzeptierendes Blatt die Tiefe $< p(|x|)$ hat, akzeptiert die NTM in polynomieller Zeit.

4.3.4 Inklusion

Es gilt weiterhin:

$$P \subseteq NP \subseteq EXPTIME \subseteq NEXPTIME$$

4.4 In-Place-Acceptance

Eine DTM akzeptiert ein Wort x in In-Place-Acceptance, wenn die Anzahl der Bandzellen, die die DTM besucht $< |x|$ ist. Man kann dieses Problem lösen, indem man die Bandzellen mit Hilfe einer universellen Turingmaschine mitzählt. Wenn mehr als $|x|$ Bandzellen besucht werden, verwirft die universelle Turingmaschine. Wir wollen jedoch auch mit dem Verfahren herausfinden, ob eine DTM wirklich akzeptiert und nicht in In-Place einfach endlos läuft. Um das herauszufinden, müssen wir die Konfigurationen auf dem Band und in den Zuständen betrachten. Wenn eine DTM ein und dieselbe Konfiguration zwei mal durchläuft (Band gleich und Zustand gleich) dann kann man sagen, daß sie sich in einer Endlosschleife befindet. Nun ist es naheliegend, daß man alle Situationen speichert. Dies geht jedoch nur in exponentielem Platz. Wir können In-Place-Acceptance in PSPACE realisieren, indem wir für jeden Konfigurationswechsel die Wechsel davor wiederholt generieren und schauen, ob ein doppelter dabei ist. Dabei werden wir nur konstanten Platz benötigen, welcher in PSPACE ist.

4.5 Platzkomplexität

4.5.1 $NP \subseteq PSPACE$

Da wir eine NTM mit einer DTM simlulieren können und wir bei dieser Simulation einen Berechnungsbaum erstellen, der mit einem Backtracking-Verfahren durchlaufbar ist, liegt ist $NP \subseteq PSPACE$. Das Backtrackingverfahren benötigt nur PSPACE Platz, da es der Platz durch die Tiefe des Baumes beschränkt ist.
Also:

$$NP \subseteq PSPACE \subseteq EXPTIME$$

4.5.2 Satz von Savitch: PSPACE = NPSPACE

Dies sagt der Satz von **Savitch**, welchen wir nicht bewiesen haben.

4.6 Zusammenfassung

$$P = PTIME \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$$

5 P-NP-Problem

Unter dem $P-NP$-Problem versteht man die Frage, ob $P = NP$ gilt.Da wir auf jeden Fall sagen können, daß $P \subseteq NP$ gilt, kann man das Problem auch wie folgt formuliert betrachten:

Läßt sich jedes Problem, das sich nichtdeterministisch in polynomieller Zeit lösen läßt, auch deterministisch in Polynomialzeit lösen?

Es gilt als wahrscheinlich das $P \neq NP$ gilt. (Da noch keine Algorithmen gefunden worden sind, die die obigen Fragestellungen zugunsten von $P = NP$ lösen könnten.) *Das P-NP-Problem ist das berühmteste Problem der theoretischen Informatik.*

5.1 Polynominalzeitreduktion mit Beispiel

Wenn wir beweisen, daß ein Problem in NP ist, können wir für alle anderen NP-Probleme beweisen, daß sie in NP sind. Dies geht indem wir das zu beweisende Problem auf das bekannte Problem mit Hilfe eines Algorithmusses zurückführen. Dieser Algorithmus muß in polynomieller Zeit laufen.
Definition Polynomialzeitreduktion: Seien Σ und Γ Alphabete und $L \subseteq \Sigma^*, K \subseteq \Gamma^*$. L heißt auf K in polynomieller Zeit reduzierbar, wenn es eine totale berechenbare Fkt. $f : \Sigma^* \to \Gamma^*$gibt, so daß

- $x \in L$ gdw $f(x) \in K$
- f ist in polynomieller Zeit berechenbar.

In diesem Fall schreiben wir $L \leq_{poly} K$.
Es gilt:

- $K \in P \Longrightarrow L \in P$
- $K \in NP \Longrightarrow L \in NP$
- $K \in PSPACE \Longrightarrow L \in PSPACE$

Beispiel: siehe z.B. Musterlösung 6.Übungsblatt, Aufgabe 6.3$(a) DHC \leq_{poly} TSP$

5.2 NP-Hart, NP-Vollständig

NP-vollständige Probleme sind die schwierigsten Probleme in NP. Dies erklärt sich daraus, das sich alle anderen Probleme aus NP auf die NP-vollständigen Probleme reduzieren lassen.

Definition NP-hart, NP-vollständig Sei K eine Sprache. K heißt $NP-hart$, wenn für alle Sprachen $L \in NP$ gilt

$$L \leq_{poly} K.$$

K heißt NP- vollständig, wenn K in NP liegt und NP -hart ist.

Der folgende Satz zeigt, das die NP -vollständigen Probleme tatsächlich die schwierigsten Probleme in NP sind. Er ist einer der zentralen Sätze des $P - NP$ Problems:

> Sobald für ein NP -vollständiges Problem ein deterministischer Polynomialzeitalgorithmus angegeben werden kann, dann liegen alle NP - Probleme in P.

5.3 Satz von Cook

Das erste Problem, für das die NP -Vollständigkeit bewiesen wurde, war das Erfüllbarkeitsproblem der Aussagenlogik (SAT), und ist als Satz von Cook bekannt. Der erste Beweis dafür, das ein Problem NP -vollständig ist, ist zugleich der schwierigste. Man kann nun das Prinzip der polynomiellen Reduzierbarkeit benutzen, um die NP-Vollständigkeit anderer Probleme nachzuweisen.

5.4 Beweisskizze, daß SAT (satisfiability problem) NP-Vollständig ist

Dies ist wirklich nur eine Skizze. Man kann den Beweis noch wesentlich genauer ausführen. Dies soll nur als ein roter Faden dienen.
Definition SAT: Das Erfüllbarkeitsproblem der Aussagenlogik, kurz SAT, ist das folgende:
gegeben: eine Formel F der Aussagenlogik
gefragt: Ist F erfüllbar, d.h. gibt es eine Belegung der Variablen mit Konstanten $\in 0, 1$, so daß F den Wert 1 erhält?

Um nun nachzuweisen, daß SAT NP-vollständig ist, ist folgendes zu tun:
1. SAT $\in NP$.
2. SAT ist NP-hart.
zu 1: Mit der Guess-and-Check-Methode kann man nachweisen, dass SAT $\in NP$ ist. Dazu wird zuerst festgestellt, welche Variablen in der Formel vorkommen. Dann wird nichtdeterministisch eine Belegung für die einzelnen Variablen geraten (d.h. es existieren genau 2^k mögliche Rechnungen, wobei k die Anzahl der Variablen.) Als nächstes wird deterministisch geprüft, ob F erfüllend ist, also den Wert 1 hat. Da nur k verschiedene Variablen in der Formel vorkommen, ist die nichdeterministische Rechenzeit polynomial.
zu 2: Wir müssen zeigen, daß alle Probleme, die in NP liegen, auf SAT reduzierbar sind. L ist hierzu ein beliebiges NP-Problem. Also muß gelten:

$$L \leq_{poly} SAT$$

Das Grundgerüst dieses Beweises ist, daß man eine Turingmaschine τ konstruiert, die L entscheidet. Diese Turingmaschine wird dann in eine KNF umgeformt, die von SAT gelöst wird.
Die KNF F hat die Bauart:

$$F = R \wedge A \wedge \ddot{U}_1 \wedge \ddot{U}_2 \wedge E$$

wobei

- R gewisse Randbedingungen beschreibt (Es gibt immer nur einen aktuellen Zustand und eine aktuelle Bandposition in der Turingmaschine (zu jedem Zeitpunkt t und jeder Bandposition i für genau ein a muß gelten $band_{t,i,a} = 1$)

- A die Anfangsbedingung (A beschreibt den Zustand der Variablen für den Zeitpunkt $t = 0$
- $\ddot{U}_1$ und $\ddot{U}_2$ Übergangsbedingungen
- und E Endbedingungen darstellen.

Wenn ein Eingabewort $x \in L\tau$, kann man eine erfüllende Bedingung für F aus der akzeptierenden Berechnung für τ für x konstruieren.
Wir können in polynomieller Zeit F für SAT aus der Turingmaschine τ erstellen. Damit gilt:

$$L \leq_{poly} SAT$$

5.5 NP-Vollständige Probleme

Es gibt prinzipielle Techniken für NP -Vollständigkeitsbeweise:
1: Man muß zeigen, daß das Problem $\in NP$ ist. Dazu reicht ein umgangssprachlich formulierter nichtdeterministischer Algorithmus (meist Guess-and-Check-Methode (rate und verifiziere)) aus.
2: Es ist die NP-Härte zu zeigen. Dazu bedient man sich meist des Prinzips der polynomiellen Reduktion.
Ist K das Problem, für das man die NP-Vollständigkeit nachweisen möchte, dann zeigt man

$$\text{bekanntes NP-hartes Problem L } \leq_{poly} K$$

Für die Reduktion gibt es drei wesentliche Strategien:

- **Spezialisierung:** Man zeigt, daß L ein Spezialfall von K ist. (z.B. SUBSUM $\leq_{poly}$ RP)
- **Lokale Ersetzung:** Man zerlegt die Eingabe von L in Einheiten, die dann unabhängig voneinander zu Eingabestücken für K eingesetzt werden. (z.B. SAT $\leq_{poly}$ 3SAT)
- **Transformation mit verbundenen Komponenten:** Man zerlegt die Eingabe von L in Einheiten (wie bei der Lokalen Ersetzung), die dann jedoch zu Eingabestücken für K kombiniert werden. (z.B. SAT $\leq_{poly}$ CP)

5.5.1 3SAT

Definition 3SAT (Erfüllbarkeitsproblem für 3KNF):
Gegeben: aussagenlogische Formel α in 3KNF
Gefragt: Ist α erfüllbar?

3SAT ist NP-vollständig.

Beweis indem man zeigt: SAT $\leq_{poly}$ 3SAT.

5.5.2 Cliquenproblem

Definition CP (Cliquenproblem):
Gegeben: ungerichteter Graph $G = (V, E)$ und eine natürliche Zahl k
Gefragt: Gibt es für G eine Clique der Größe k?

CP ist NP -vollständig.

Beweis indem man zeigt: 3SAT $\leq_{poly}$ CP.

5.5.3 Rucksackproblem

Definition RP (Rucksackproblem):
Gegeben: natürliche Zahlen $p_1, ..., p_n, w_1, ..., w_n, q, K$
Gefragt: Gibt es eine Teilmenge I von $\{1, ..., n\}$ mit $\Sigma_{i \in I} w_i \leq K$ und $\Sigma_{i \in I} p_i = q$?

RP ist NP -vollständig

Beweis indem man die NP -vollständigkeit des Spezialfalls SUBSUM (Teilsummenproblem) zeigt.

5.5.4 0/1 Integer Linear Programming

Definition 0/1 Integer Linear Programming (0/1 ILP):
Gegeben: Lineares Ungleichungssystem $\mathbf{AX} \leq \mathbf{b}$ mit ganzzahligen Koeffizienten
Gefragt: Gibt es einen Lösungsvektor $\mathbf{x}$, dessen Komponenten 0 oder 1 sind?

0/1 ILP ist NP -vollständig.

Beweis indem man zeigt 3SAT $\leq_{poly}$ 0/1 ILP.

5.6 coNP

Die Antworten NEIN und JA sind für deterministische Entscheidungsverfahren symmetrisch. (d.h. sie können vertauscht werden, um die Komplementsprache zu entscheien) Für nichtdeterministische Entscheidungsverfahren ist dies nicht so. Man sagt, die die Antworten JA und NEIN sind asymmetrisch. Dennoch ist das komplementäre Problem entscheidbar. Die Klasse P ist unter der Komplementbildung abgeschlossen. Entsprechendes gilt für jede deterministische Komplexitätsklasse (d.h., für jede Sprache $L \in P$ gilt auch $\bar{L} \in P$).

Definition Komplementkomplexitätsklassen: Sei $\mathcal{C}$ eine Komplexitätsklasse. Die Klasse *co*$\mathcal{C}$ besteht aus allen Sprachen L, deren Komplement $\bar{L}$ in $\mathcal{C}$ liegt.
Es gilt also

$$P = coP \text{ und } PSPACE = coPSPACE.$$

Die Klasse $coNP$ besteht also aus allen Sprachen L, deren Komplement $\bar{L}$ in NP liegt. Der zusammenhang zwischen NP und $coNP$ ist bisher noch ungeklärt. Man vermutet, daß $NP \neq coNP$. Aus NPC $\cap$ coNP $\neq \oslash$ folgt $NP = coNP$.

5.7 PSPACE - Vollständigkeit

Die Frage, ob $NP = PSPACE$, ist noch nicht geklärt. Man vermutet, daß $PSPACE \subset NP$.
Definition PSPACE-Vollständigkeit, PSPACE-Härte: Sei K eine Sprache. K heißt $PSPACE$ -vollständig, wenn

- $K \in PSPACE$ und
- jede Sprache $L \in PSPACE$ auf K polynomiell reduzierbar ist.

Mit den bisherigen wissen, daß PSPACE wahrscheinlich eine echte Obermenge von NP ist, und diese wiederum wahrscheinlich echte Obermenge von P, können wir folgenden Schluß ziehen:

Sobald für ein $PSPACE$ -vollständiges Problem ein (deterministischer) Polynomialzeitalgorithmus angegeben werden kann, dann liegen alle $PSPACE$ -Probleme in P .

6 Grammatiken

6.1 Clomsky Hierarchie

6.1.1 Typ 0 Grammatik

In der Typ 0 Grammatik ist jede regel erlaubt. Es kann sein, daß das Wort, welches durch die Grammatik gebildet wird, zuerst zunimmt und dann wieder abnimmt. Grammatiken vom Typ 0 sind ein weiterer Formalismus für Semientscheidbarkeit. Da die Klasse der Sprachen vom Typ 0 mit der Klasse der von Turingmaschinen akzeptierten Sprachen übereinstimmt, ist klar, daß für Sprachen vom Typ 0 das Wortproblem unentscheidbar ist.

6.1.2 Typ 1 Grammatik - Kontextsensitive Grammatik

Für die Kontextfreie Grammatik gilt stets für $u \to v$:

$$|u| \leq |v|$$

Das heißt also, daß die linke Seite einer Produktion kleiner gleich sein muß als die rechte Seite einer Produktion. Es geht also nicht, daß Wörter wieder abnehmen. Für Sprachen vom Typ 1 ist das Wortproblem PSPACE -vollständig.

6.1.3 ϵ-Sonderregel der Kontextsensitiven Grammatik

Wir laufen bei der Typ 1 Grammatik in das Problem, daß die Regel $S \to \epsilon$ nicht erlaubt ist, da die linke Seite eins lang ist, die rechte Seite aber leer ist. Wir wollen aber trotzdem dies zulassen, deshalb definieren wir die ϵ-Regel als Sonderfall:

1. Die Regel $S \to \epsilon$
2. Für alle übrigen Regeln gilt die Bedingung wie oben genannt. S kommt auf der rechten Seite in diesen nicht mehr vor.

6.1.4 Typ 2 Grammatik - Kontextfreie Grammatik

Für eine kontextfreie Grammatik wird ein Non-Terminal auf der linken Seite der Produktion benötigt. Auf der rechten Seite kann stehen, was will. Jede kontextfreie Grammatik ist in eine kontextsensitive umformbar, auch wenn noch weitere ϵs später auftauchen, können diese mit Hilfe der Konstruktion einer äquivalenten ϵ-freien CFG umgewandelt werden. Das Wortproblem für Sprachen vom Typ 2 ist in kubischer Laufzeit

(CYK-Algorithmus) lösbar.
Konstruktion einer äquivalenten ϵ-freien CFG: Es ist möglich jede CFG in eine ϵ-freie CFG umzuwandeln. Jede ϵ-freie CFG ist kontextsensitiv(Typ1). Die erreicht man durch ausführen folgender Schritte:

- Man sucht alle Nichtterminale, mit denen man durch irgendwelche erlaubten Produktionen zum ableiten des leeren Wortes gelangt, und markiert diese.
- Dann werden alle ϵ-Regeln aus der Grammatik entfernt.
- Als letztes sucht man solche Produktionen, in denen ein markiertes Nichtterminal auf der rechten Seite vorkommt, umgeben von mindestens einem weiteren Terminal oder Nichtterminal, und fügt diese Produktionen, sofern noch nicht vorhanden, neu in das Produktionssystem bei, die durch weglassen der markierten Nichtterminale entstehen würden. Es müssen dabei alle Möglichkeiten berücksichtigt werden.

6.1.5 Typ 3 Grammatik - reguläre Grammatik

Bei einer Regulären Grammatik steht ein einzelnes Non-Terminal links. Außerdem steht rechts entweder

- ein ϵ,
- ein Terminal oder
- ein Terminal gefolgt von einem Non-Terminal.

Anderes ist nicht erlaubt. Das Wortproblem für Sprachen vom Typ 3 kann in linearer Zeit gelöst werden.

6.1.6 Inklusion

Diese Inklusion ist auch als Clomsky Hierarchie bekannt. Es ist bewiesen, daß die Sprache vom Tpy i-1 die Sprache vom Typ i enthält.

$$Typ\,3 \subset Typ\,2 \subset Typ\,1 \subset Typ\,0$$

6.2 Eigenschaften von Grammatiken

allgemeines: Die Herleitung von Wörtern aus Grammatiken ist ein nichtdeterministischer Prozeß, der mit NTMs und damit auch mit DTMs vollzogen werden kann. Wir können also folgendes sagen:

- Zu jeder Grammatik G gibt es eine NTM $\mathcal{T}$ mit $\mathcal{L}(G) = \mathcal{L}(\mathcal{T})$
- Zu jeder DTM $\mathcal{T}$ gibt es eine Grammatik G mit $\mathcal{L}(\mathcal{T}) = \mathcal{L}(G)$

6.2.1 Abschlußeigenschaften

Die Abschlußeigenschaften der einzelnen Sprachen für die wichtigsten Verknüpfungsoperatoren *Vereinigung, Durchschnitt, Komplement, Konkatenation und Kleeneabschluß* sind wie folgt:

	Vereinigung	Durchschnitt	Komplement	Konkatenation	Kleeneabschluß
Typ0-Sprachen	ja	ja	nein	ja	ja
Typ1-Sprachen	ja	ja	ja	ja	ja
Typ2-Sprachen	ja	nein	nein	ja	ja
Typ3-Sprachen	ja	ja	ja	ja	ja

6.3 Kontextsensitve Sprachen

Kontextabhängige Eigenschaften spielen bei der semantischen Analyse eines Übersetzers eine wichtige Rolle. Kontextsensitive Sprachen können mit einer Unterklasse von Turingmaschinen, die mit linearen Platz (Linear beschränkte Automaten = LBAs) auskommen, dargestellt werden.

6.4 Linear beschränkte Automaten (LBAs)

LBAs sind NTMs, die mit zwei Begrenzungssymbolen arbeiten, die jeweils das rechte bzw. linke Ende des nutzbaren Bandbereichs kennzeichnen. Durch die Begrenzungen können LBAs nie mehr als n Zeichen speichern, wenn n die Eingabelänge ist. Bei einer CSG ist es ebenfalls aus symmetrischen Gründen so, daß die Wortlänge höchstens |w| ist. Man kann also sagen, daß die durch eine CSG erzeugten Sprachen L genau die sind, die von einem LBA akzeptiert werden. Die Frage, ob deterministische LBAs genauso Mächtig wie nichtdeterministische LBAs sind, ist noch ungeklärt.

6.4.1 LBA-Acceptance

Gegeben: LBA T mit dem Eingabewort x
Gefragt: Gilt $x \in L_{LBA}(T)$?

LBA-Acceptance ist PSPACE -vollständig.

7 Reguläre Sprachen

Das Konzept regulärer Sprachen und verwandter Formalismen, wie z.B. endliche Automaten, findet im Bereich der lexikalischen Analyse beim Compilieren höherer Programmiersprachen anwendung. Endliche Automaten können als Sprachakzeptoren benutzt werden. Das Wortproblem ist in linearer Zeit lösbar. Sie sind eine sehr eingeschränkte Variante von Turingmaschinen bei der nur ein Band zur verfügung steht. Außerdem darf der Lesekopf nur nach rechts bewegt werden. Das einzige zur verfügung stehende Speichermedium sind die Zustände.

7.1 Deterministische endliche Automaten (DFA)

Definition Deterministischer endlicher Automat(DFA): Ein DFA ist ein Tupel

$$\mathcal{M} = (\mathcal{Q}, \Sigma, \delta, q_0, F)$$

bestehend aus

- einer endlichen Menge $\mathcal{Q}$ von *Zuständen*
- einem endlichem *Alphabet* Σ
- einer *partiellen Funktion* (auch Übergangsfunktion genannt) $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$
- einem *Anfangszustand* (auch Startzustand genannt) $q_0 \in \mathcal{Q}$
- einer Menge $F \subseteq \mathcal{Q}$ von *Endzuständen* (auch Akzeptanzzustände genannt)

Jede Berechnung, in welcher der Automat anhält, ohne das Eingabewort zu Ende gelesen zu haben, ist verwerfend.
DFAs und reguläre Grammatiken haben dieselbe Ausdrucksstärke. Daher gilt:

Zu jedem DFA gibt es eine reguläre Grammatik, die dieselben Sprachen akkzeptieren.

7.2 Nichtdeterministische endliche Automaten (NFA)

Es gibt zwei Unterschiede zu DFAs:
1. es kann mehrere Anfangszustände geben
2. jeder Zustand hat eine Menge von möglichen Folgezuständen

Definition Nichtdeterministischer endlicher Automat (NFA) Ein NFA ist ein Tupel

$$\mathcal{M} = (\mathcal{Q}, \Sigma, \delta, \mathcal{Q}_0, F)$$

bestehend aus einer endlichen Menge $\mathcal{Q}$ von Zuständen, einem endlichen Alphabet Σ, einer Menge $F \subseteq Q$ von Endzuständen und

- einer totalen *Übergangsfunktion* $\delta : \mathcal{Q} \times \Sigma \to 2^{\mathcal{Q}}$
- einer Menge $\mathcal{Q}_0 \subseteq \mathcal{Q}$ von Anfangszuständen

Im Gegensatz zu DFAs kann ein Wort w viele Läufe in einem NFA haben. Für die Akzeptanz wird lediglich gefordert, daß einer der Läufe für w akzeptierend ist.
NFAs mit ϵ-Übergängen: ϵ-Übergänge sind spontane Zustandsveränderungen, die unabhängig vom gelesenen Zeichen stattfinden, und die Position des Lesekopfs unverändert lassen. Zu jedem ϵ erweiterten NFA gibt es einen NFA ohne ϵ-Übergängen, der dieselbe Sprache akzeptiert.

7.3 Äquivalenz von DFAs und NFAs

Jeder NFA kann als DFA aufgefaßt werden (Potenzmengen Konstruktion - es ensteht ein exponentiell großer DFA in exponentieller Zeit). Natürlich kann man auch ein DFA in einen NFA umwandeln.
Es gilt also:

$$\text{DFA} \implies \text{reguläre Grammatik} \implies \text{NFA} \implies \text{DFA}$$

7.4 Algorithmen für den Nachweis von Eigenschaften regulärer Sprachen

Das Wortproblem: Ist $\mathcal{M}$ ein DFA mit dem Alphabet Σ und$w$ ein Wort über Σ, dann kann die Frage

$$\text{Gilt } w \in \mathcal{L}(\mathcal{M})$$

in linearer Zeit $O(|w|)$ beantwortet werden.

Äquivalenzproblem: *Gegeben:* zwei DFAs $\mathcal{M}_1$ und $\mathcal{M}_2$ mit demselben Eingabealphabet.
Gefragt: gilt $\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_2)$. In $O(|\mathcal{Q}_1|*|\mathcal{Q}_2|*|\Sigma|)$ lösbar.

Leerheitsproblem: *Gegeben:* ist ein DFA $\mathcal{M}$.
Gefragt: gilt $\mathcal{L}(\mathcal{M}) = \emptyset$. In $O(|\mathcal{Q}| *|\Sigma|)$ lösbar.

Endlichkeitsproblem: *Gegeben:* Ein DFA $\mathcal{M}$.
Gefragt: gilt $|\mathcal{L}(\mathcal{M})|<\infty$. In $O(|\mathcal{Q}| *|\Sigma|)$ lösbar.

7.5 Das Pumping-Lemma für reguläre Sprachen

Das Pumping Lemma präsentiert ein notwendiges Kriterium für reguläre Sprachen und kann für den Nachweis genutzt werden, daß eine Sprache nicht regulär ist.
Definition Pumping Lemma für reguläre Sprachen: Sei $L \subseteq \Sigma^*$ eine reguläre Sprache. Dann gibt es eine ganze Zahl n≥1, so daß jedes Wort $x \in L$ mit $|x| \geq n$ wie folgt zerlegt werden kann:
$x = uvw$ mit Wörtern $u, v, w \in \Sigma^*$, so daß

- $|v| \geq 1$
- $|uv| \leq n$
- $uv^k w \in L$ für alle $k \in IN$

7.6 Reguläre Ausdrücke

Reguläre Ausdrücke sind ein Formalismus, der sehr intuitive Schreibweisen für reguläre Sprachen ermöglicht.
Definition Syntax regulärer Ausdrücke: Sei Σ ein Alphabet. Die Menge der regulären Ausdrücke über Σ ist durch folgende induktive Definition gegeben:

- $\emptyset$ und ϵ sind reguläre Ausdrücke.
- Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck.
- Mit α und β sind auch $(\alpha\beta)$,$(\alpha + \beta)$ und (α^*) reguläre Ausdrücke.
- Nichts sonst ist ein regulärer Ausdruck.

Zu jedem regulärem Ausdruck gibt es einen NFA, der dieselbe Sprache akzeptiert.Man kann z.B. NFAs mit ϵ -übergängen benutzen. Außerdem kann man aus jedem DFA eien regulären Ausdruck bilden, die ebenfalls dieselbe Sprach akzeptieren. Dazu bedient man sich der Methode des dynamischen Programmierens.

7.7 Syntaxdiagramme

Syntaxdiagramme sind eine graphische Darstellungsform der Regeln, die zur Bildung von Grundsymbolen zugelassen sind. Jeder reguläre Ausdruck $\alpha \neq \emptyset$ läßt sich graphisch durch ein Syntaxdiagramm darstellen. Das Konstruktionsverfahren benutzt strukturelle Induktion über den syntaktischen Aufbau des Ausdrucks α.

Um Syntaxdiagramme in endliche Automaten zu überführen, benutzt man eine Graphentheoretische Sicht. Wir fassen ein Syntaxdiagramm als Digraphen auf, dessen Knoten mit einem Symbol $a \in \Sigma$ markiert sind. Weiter werden zwei zusätzliche unmarkierte Knoten benutzt, die für den Eingang bzw. Ausgang stehen.(start und stop)
Um ein Syntaxdiagramm in einen NFA überzuführen, bildet man einen zum Syntaxdiagramm dualen Graphen. D.h. man beschriftet die Knoten(=Zustände) des NFA mit den Kanten des Syntaxdiagramm, und die Knoten des Syntaxdiagramms werden zur beschriftung der Kanten im NFA benutzt. Der so entstandene NFA kann durch die Potenzmengenkonstruktion wieder in einen DFA umgewandelt werden.
Es gelten also folgende Transformationen:
DFA $\Longrightarrow$ regulärer Ausdruck $\Longrightarrow$ NFA $\Longrightarrow$ DFA.
Mit den oben kennengelernten Transformationen gilt:
Syntaxdiagramm $\Longrightarrow$ NFA oder DFA $\Longrightarrow$ reguläre Grammatik $\Longrightarrow$ NFA $\Longrightarrow$ regulärer Ausdruck $\Longrightarrow$ Syntaxdiagramm.

7.8 Minimierung endlicher Automaten

Unter einem minimalen DFA versteht man einen äquivalenten DFA, der die wenigsten Zustände unter allen äquivalenten DFAs hat. Der Satz von Myhill und Nerode stellt eine weitere Charakterisierung regulärer Sprachen dar und bietet die Basis für den Minimierungsalgorithmus.
Vorüberlegungen: Man kann jeder Sprache L eine Äquivalenzrelation $\sim_L$ auf Σ^* zuordnen.

Definition $\sim_L$: Es gilt x $\sim_L$ y genau dann, wenn für alle Wörter $z \in \Sigma^*$ gilt:

$$xz \in L \Leftrightarrow yz \in L$$

Zwei Wörter x und y sind also genau dann aquivalent, wenn sie sich bei Anfügen von beliebigen Wörtern z (da auch z= ϵ möglich, gilt insbesondere $x \in L \Leftrightarrow y \in L$) bzgl. der Mitgliedschaft in L gleich verhalten.

Definition $\sim_M$: Sei $M = (\mathcal{Q}, \Sigma, \delta, q_0, F)$ ein DFA. Die Äquivalenzrelation $\sim_M$ ist durch

$$x \sim_M y \text{ gdw } \delta(q_0, x) = \delta(q_0, y)$$

definiert. Also falls die Eingaben x und y den Automaten in dieselben Zustände überführt.

Es gilt $\sim_M \subseteq \sim_L = \sim_{M_0}$, wobei M_0 der Minimalautomat ist. Der Minimalautomat ist bis auf Isomorphie, d.h. umbenennen der Zustände, eindeutig bestimmt. (gilt nur für DFAs)

7.8.1 Der Satz von Myhill und Nerode

Definition Satz von Myhill und Nerode: Sei L eine Sprache.

L ist genau dann regulär, wenn der Index von L endlich ist.

Der Index ist die Anzahl der erzeugten Äquivalenzklassen.

7.8.2 Minimierungsalgorithmus:

- Im ersten Schritt erstellt man eine Tabelle aller Zustandspaare, so daß alle Knotenpaare genau einmal in dieser Tabelle erfasst werden.
- Es werden dann alle Zustandspaare markiert, von denen ein Zustand ein Endzustand ist, und der andere nicht.
- Zum Schluß nimmt man sich ein noch unmarkiertes Zustandspaar aus der Tabelle, und prüft mit einem beliebigen Zeichen aus Σ, in welches neue Zustandspaar man diese zwei ausgewählten Zustände überführen kann. Für dieses neue Zustandspaar schaut man in der Tabelle nach, ob diese bereits markiert sind. Wenn ja, dann markiere auch das anfangs ausgewählte Zustandspaar.
- Wiederhole den letzten Schritt, bis sich keine Änderungen in der Tabelle mehr ergeben
- Alle noch unmarkierten Zustandspaare kann man jeweils zu einem Zustand zerschmelzen. Zusammen mit den anderen Zuständen, die man nicht weiter zusammenfassen konnte, kann man dann den Minimalautomaten erstellen.

$\Longrightarrow$ **Laufzeit:** $O(|\mathcal{Q}^2| * |\Sigma|)$

8 Kontextfreie Sprachen

8.0.3 allgemeines:

Sprachen vom Typ0 oder Typ1 scheiden für das Wortproblem wegen unentscheidbarkeit, bzw. zeitlicher inneffiezients außer betracht.Für geklammerte Sprachstrukturen, wie sie bei höheren Programmiersprachen vorkommen, sind Sprachen vom Typ3 zu schwach. Somit bleiben für die Syntaxanalyse nur noch die Typ2 Sprachen von der Chomsky Hierarchie übrig.

8.1 Rechts- und Linksableitung

Bei einer Rechtsableitung wird immer das rechteste Nichtterminal abgeleitet. Bei einer Linksableitung wird dementsprechend das linkeste Nichtterminal abgeleitet. Es ist möglich, für ein Wort mehrere Ableitungsbäume zu konstruieren. Die Grammatik heißt dann mehrdeutig, sonst eindeutig. Eine Sprache heißt inhärent mehrdeutig, falls es keine eindeutige CFG für diese Sprache gibt. Aus einer mehrdeutigen Grammatik folgt nicht, daß die durch die Grammatik erzeugte Sprache inhärent mehrdeutig ist. (da es eine zu der Grammatik eindeutige äquivalente Grammatik geben kann)

8.2 Die Chomsky Normalform(CNF)

Eine der wichtigsten Normalformen für CFGs ist die Chomsky Normalform (CNF)
Definition Chomsky Normalform: Sei $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{P}, \mathcal{S})$ eine CFG. $\mathcal{G}$ heißt in CNF, falls eine Produktion in G von der Form

- A $\rightarrow$ a für ein $a \in \Sigma$
- oder A $\rightarrow$ BC für Nichtterminale B,C

Ableitungsbäume für solche Grammatiken sind -bis auf den letzten Ableitungsschritt- Binärbäume. Ein Wort w, das aus der durch die Grammatik erzeugten Sprache erzeugt wird, benötigt genau 2|w|-1 Ableitungsschritte.

8.2.1 Konstruktion einer CNF-Grammatik

Um eine Grammatik in die CNF zu bringen, geht man wie folgt vor:

- Zuerst entfernt man alle ϵ-Regeln aus der Grammatik
- Danach entfernt man alle Kettenregeln und prüft, ob S auf der rechten Seite der Produktionen vorkommt. Falls ja, wird ein neues Startsymbol S' mit den entsprechenden ableitungen eingeführt.
- Nun wird für jedes Terminal ein neues Nichtterminal in das Produktionssystem beigefügt, von dem man genau dieses terminal ableiten kann.
- im letzten Schritt werden die Produktionen in die benötigte Form zerstückelt. Dies kann man z.B. dadurch erreichen, indem man neue Nichtterminale einfügt.

8.2.2 Die Größe der konstruierten CNF-Grammatik

Die Größe einer CFG G wird an der Anzahl an Nichtterminalen und der Summe der Längen aller Produktionen gemessen und mit $size(G)$ bezeichnet. Mit der eben beschriebenen Methode zur erstellung einer äquivalenten CNF-Grammatik G' gilt

$$(sizeG') = O(size(G)^2) \text{ falls CFG } \epsilon\text{-frei ist}$$

Die entfernung der Kettenregel kann die Größe der Grammatik quadratisch vergrößern.

8.3 Der Cocke-Younger-Kasami(CYK) Algorithmus

Wenn eine Grammatik in CNF gegeben ist, dann läßt sich das Wortproblem für kontextfreie Sprachen durch den CYK-Algorithmus in Zeit O(n^3) und Platz O(n^2) lösen. Wenn ein Wort der Länge 1 abgeleitet wird, kann dies nur durch eine Regel der Form $A \to a$ geschehen. Für Wörter die mind. eine Länge von 2 haben, muß zunächst eine Regel der Form $A \to BC$ angewandt worden sein, und anschließend muß von B ein Anfangsstück des Wortes, und von C ein Endstück des Wortes abgeleitet worden sein. Damit ist es möglich, das Wortproblem auf zwei Teilwörter aufzuteilen, dem Anfangstück und dem Endstück des Wortes. Durch die Methode des dynamischen programmierens untersucht man nun alle Teilwörter beginnend, bei der Länge eins, und guckt, aus welchen Nichtterminalen dieses Teilwort abgeleitet worden sein kann. Diese Information legt man in einer Tabelle ab. Wenn nun ein Teilwort untersucht werden soll, so liegt diese Information evtl. schon in der Tabelle vor. Das Wort liegt in der Sprache, wenn es sich aus dem Startsysmbol ableiten lässt.

8.4 Das Pumping Lemma für kontextfreie Sprachen

Das Pumping Lemma für kontextfreie Sprachen ist ein notwendiges Kriterium, das für den Nachweis, daß eine Sprache nicht kontextfrei ist, hilfreich sein.
Definition Pumping Lemma für kontextfreie Sprachen: Sei L eine kontextfreie Sprache. Dann gibt es eine natürliche Zahl n, so daß sich jedes Wort $z \in L$ der Länge $\geq n$ wie folgt zerlegen läßt: $z = uvwxy$, wobei

- $|vx| \geq 1$
- $|vwx| \leq n$
- $uv^k wx^k y \in L$ für alle $k \in IN$

8.5 Die Greibach Normalform

Definition Greibach Normalform: Sei G eine CFG. G ist in Greibach Normalform, falls alle Produktionen in G von der Form

$$A \rightarrow aB_1B_2...B_k$$

sind, wobei $a \in \Sigma, B_1, B_2, ..., B_k \in V$ und $k \in IN$.(k=0 ist zulässig)

8.6 Kellerautomaten

Das Modell eines endlichen Automaten wird um einen Speicher erweitert, so daß genau die kontextfreien Sprachen durch einen solchen Automaten erkannt werden können. Dazu wird das NFA-Modell durch einen Keller(=Stack) erweitert. Die möglichen Aktionen eines Kellerautomaten hängen jetzt nicht nur vom Zustand und gelesenem Zeichen ab, sondern auch vom obersten Kellerinhalt.
Definition nichtdeterministischer Kellerautomat: Ein NKA ist ein Tupel

$$\mathcal{K} = (\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, \#, \mathcal{F})$$

bestehend aus

- einer endlichen Menge $\mathcal{Q}$ von Zuständen
- einem Eingabealphabet Σ
- einem Kelleralphabet Γ
- einem Anfangszustand $q_0 \in \mathcal{Q}$
- einem Kellerstartsymbol $\#$
- einer Menge $\mathcal{F} \subseteq \mathcal{Q}$ von Endzuständen
- einer Übergangsfunktion $\delta : \mathcal{Q} \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow 2^{\mathcal{Q} \times \Gamma^*}$

8.6.1 Konfigurationen und Konfigurationswechsel

Eine Konfiguration für einen NKA ist ein Tripel

$$\mathrm{K} = (q, x, \xi)$$

bestehend aus einem Zustand, einem Wort aus Σ^* und einem Wort ξ aus Γ
Intuitiv bedeutet

$$\delta(q, a, A) \ni < p, BC >$$

folgendes: Wenn sich der Kellerautomat in Zustand q befindet, das Eingabezeichen a liest und A das oberste Kellerzeichen ist, so kann der Kellerautomat im nächsten Schritt in den Zustand p wechseln und das oberste Kellerelement durch BC ersetzen. Es gibt auch sog. spontane Übergänge, bei denen nichts vom Eingabeband gelesen wird.

8.6.2 Akzeptanzverhalten

Man kann Kellerautomaten durch leeren Keller, oder durch Akzeptantzustände akzeptieren lassen. Die beiden Arten von Akzeptanzbedingungen sind äquivalent.

8.6.3 Äquivalenz von NKAs und kontextfreien Grammatiken

Wenn eine CFG in Greibach-Normalform vorliegt, kann der NKA den Ableitungsprozeß simulieren. In jedem Schritt steht der Kellerinhalt für die Folge der noch zu ersetzenden Nichtterminale. Das Anwenden einer Regel A $\rightarrow$ a$B_1, ..., B_n$, wobei A das aktuell oberste Kellersymbol ist, wird so simuliert:

- A wird vom Keller entfernt,
- $B_1, .., B_n$ werden in umgekehrter Reihenfolge auf den Keller gelegt.

Wenn A $\rightarrow$ a angewandt wird, dann wird lediglich A aus dem Keller entfernt.
Wir erhalten bisher: Sei L eine Sprache. Dann sind folgende Aussagen äquivalent:

- L ist kontextfrei, d.h. $L = \mathcal{L}(G)$ für eine CFG G
- $L = \mathcal{L}(\mathcal{K})$ für einen NKA $\mathcal{K}$
- $L = \mathcal{L}_\epsilon(\mathcal{K})$ für einen NKA $\mathcal{K}$

8.6.4 NKAs mit zwei Kellern

Durch Hinzunahme ändert sich die Leistung eines Kellerautomaten. Ein Kellerautomat mit zwei Kellern hat die Mächtigkeit von Turingmaschinen, was wiederum fatale Folgen für das Wortproblem hat.(Unentscheidbarkeit)

8.6.5 Durchschnitt zwischen kontextfreien und regulären Sprachen

Ist L_1 eine reguläre und L_2 eine kontextfreie Sprache, so ist $L_1 \cap L_2$ kontextfrei.

9 Deterministisch kontextfreie Sprachen

Für die Syntaktische Analys ist eine echte Unterklasse der kontextfreien Sprachen ausreichend, für die das Wortproblem in deterministischer Zeit lösbar ist. Diese Unterklasse entspricht den deterministischen Kellerautomaten(DKAs). Deterministische Kellerautomaten sind NKAs, für denen der jeweils nächste Schritt eindeutig festgelegt ist. Die ϵ-Übergänge werden beibehalten.
Deterministischer Kellerautomat (DKA): Ein DKA ist ein NKA

$$\mathcal{K} = (\Pi, \Sigma, \Gamma, \delta, q_0, \#, \mathcal{F}),$$

so daß für alle $q \in \mathcal{Q}, a \in \Sigma$ und $A \in \Gamma$ gilt:

- $|\delta(q, a, A)| \leq 1$
- $|\delta(q, \epsilon, A)| \leq 1$
- Aus $\delta(q, \epsilon, A) \neq \emptyset$ folgt $\delta(q, a, A) = \emptyset$

9.1 Akzeptanzverhalten

Im Gegensatz zu NKAs sind die beiden Akzeptanzvarianten Akzeptanz durch Endzustände und Akzeptanz durch leeren Keller für DKAs nicht gleichwertig. Die Akzeptanz durch leeren Keller ist für DKAs schwächer als die Akzeptanz durch Endzustände. Es gilt: kontextfreie Sprachen $\supset$ deterministische kontextfreie Sprachen $\supset$ reguläre Sprachen

9.1.1 Präfixeigenschaft

Definition Präfixeigenschaft: Sei $L \subseteq \Sigma^*$. L hat die Präfixeigenschaft, wenn für alle Wörter $w \in L$ gilt: Ist x ein echtes Präfix von w, so ist $x \notin L$.
DKAs mit der Akzeptanz durch leeren Keller können genau diejenigen deterministischen kontextfreien Sprachen erkennen, die die Präfixeigenschaft haben.

9.1.2 Abschlußeigenschaften

	Vereinigung	Durchschnitt	Komplement	Konkatenation	Kleeneabschluß
det. kontextfreie Sprachen	nein	nein	ja	nein	nein

9.1.3 Durchschnitt zwischen det. kontextfreien und regulären Sprachen

Ist L_1 regulär und L_2 det. kontextfrei, dann ist $L_1 \cap L_2$ det. kontextfrei.

9.2 LR(0)-Grammatiken

Für LR(k)-Grammatiken ist das Wortproblem in Linearzeit lösbar. Die Bezeichnung LR(k) erklärt sich daraus, daß für das Wortproblem

L: die Eingabe von links nach rechts gelesen wird

R: eine Rechtsableitung in Bottom-Up Manier konstruiert wird

k: mit einem Lookahead der Länge k gearbeitet wird

Das Lookahead steht für das Präfix des noch unbearbeiteten Teil der Eingabe. Es wird immer ein Zeichenkette der Länge k zu dem bereits bearbeiteten Teilwort hinzugenommen, und geprüft, ob sich aus den Produktionen eine Reduktion durch Rechtsableitungen zu der gelesenen Teilwortes ableiten läßt. Falls ja wird das nächste Teilwort der Länge k geprüft, falls nein, dann verwirft der Parser das Wort. Der Parser akzeptiert ein syntaktisch korrektes Programm, falls das Eingabe Wort aus S ableitbar ist. Für die LR(0)-Grammatik wird weiter gefordert, das sie keine nutzlosen Variablen enthält und das Startsymbol in keiner Regel auf der rechten Seite vorkommt.

9.2.1 Griff

Ein Griff ist die Menge der Terminale, die im letzten Rechtsableitungsschritt gebildet worden sind.

9.2.2 Rechtssatzform

Alle Wörter die aus dem Startsymbol S durch eine Rechtsableitung herleitbar sin, nennt man Rechtssatzform.