Vorlesung Theoretische Informatik I
Greifswald, Sommersemester 1995

Inhalt:

1. Vorbemerkungen

Für die **vertiefende informationstechnische Bildung** in Form der Informatik werden im Gesamtkonzept der Bund-Länder-Kommission für Bildungsplanung und Forschungsförderung (BLK) folgende Aufgaben genannt:

- Behandlung der Wirkungsweise, Leistungsfähigkeit und Leistungsgrenzen von Computern
- Vermittlung von Problemlösemethoden
- Vermittlung von Kenntnissen bestimmter Programmiersprachen
- Behandlung des strukturierten Programmierens und der Datenstrukturen
- Einsatz von Computern für Berechnungen, für die Erstellung von Grafiken und für die Simulation von Verfahren
- Erörterung von Prozeßsteuerung durch Mikroprozessoren

Bei der praktischen Umsetzung des BLK-Konzepts geht jedes Bundesland seinen eigenen Weg. Das Spektrum der angebotenen Kurse reicht z. B. an Gymnasien vom einjährigen Grundkurs über Grund-kurse mit Abiturprüfung bis hin zum dreijährigen Leistungskurs. Der Schwerpunkt liegt bei

den meisten Lehrplänen beim algorithmischen Problemlösen, also im Bereich der praktischen und angewandten Informatik. Ohne ein gewisses Minimum an theoretischen Kenntnissen bleibt aber der Zugang zu wesentlichen Aussagen der Informatik, die heute schon allgemeinbildende Bedeutung haben, versperrt. Das betrifft insbesondere Fragen der Komplexität von Algorithmen und der damit verbundenen Leistungsfähigkeit und Leistungsgrenzen des Computers. Falsche Meinungen derart, daß auch das komplizierteste Problem mit einem passenden Programm und einem hinreichend großen und schnellen Computer gelöst werden kann, dürfen nicht entstehen oder gar gefördert werden. Diese Gefahr besteht aber immer bei einer betont empirisch-pragmatischen Herangehensweise ohne Rückgriff auf eine theoretische Basis.
Die Vorlesung Theoretische Informatik I hat deshalb das Ziel, die Lehramtsstudenten in die Theorie der Automaten, Sprachen und Algorithmen einzuführen, die für das Verständnis zahlreicher Anwendungen wichtig sind.

Die Theoretische Informatik umfaßt folgende Teilgebiete:
Berechnungstheorie
Komplexitätstheorie
Automatentheorie
Theorie der formalen Sprachen

In dem weiterführenden Kurs Theoretische Informatik II erfolgt später zu ausgewählten Themen eine Vertiefung.

2. Algorithmen

Viele Jahrhunderte, und zwar so lange wie es nur darum ging, für bestimmte Probleme geeignete Lösungsalgorithmen zu finden, gaben sich die Mathematiker mit dem folgenden anschaulichen Algorithmusbegriff zufrieden:

Definition (Algorithmus)
Ein **Algorithmus** ist ein als Folge von Anweisungen beschriebenes, schrittweise ablaufendes Verfahren zur Lösung eines Problems, das folgenden Bedingungen genügt:

- **Endlichkeit**, d. h. die Anweisungsfolge ist durch einen endlichen Text beschreibbar
- **Determiniertheit**, d. h. an jeder Stelle ist der Ablauf der Anweisungen eindeutig festgelegt
- **Universalität**, d.h. das Verfahren gilt für die Lösung einer ganzen Klasse von Aufgaben, nicht nur für eine spezielle
- **Ausführbarkeit**, d. h. die Anweisungen sind für den Befehlempfänger (Mensch oder Maschine) verständlich formuliert und für diesen ausführbar

Häufig findet man außerdem die Forderung nach

- **Terminiertheit**, d. h. nach endlich vielen Schritten liefert die Anweisungsfolge eine Lösung des gestellten Problems.

Beispiele für Algorithmen sind die Vorschriften zum Addieren, Subtrahieren und Multiplizieren von natürlichen Zahlen oder der euklidische Algorithmus zur Bestimmung des größten gemeinsamen Teilers (ggT) zweier natürlicher Zahlen. Übliche Kochrezepte, Bastelanleitungen, Spielregeln u. ä. werden häufig auch zu den Algorithmen gezählt, sind aber mit Vorsicht zu genießen, da sie nicht selten Teile enthalten, die unterschiedlich interpretiert werden.
Ein Algorithmus beschreibt also eine Funktion $\mathbf{f : E \rightarrow A}$ von der Menge E der Eingabedaten in die Menge A der möglichen Ausgabedaten. Ist f eventuell nicht überall in E definiert, dann ist f eine **partielle Funktion**. Überall definierte Funktionen sind **totale Funktionen**.

2. 1. Berechenbarkeit, Entscheidbarkeit und Aufzählbarkeit im anschaulichen Sinne

Mit Hilfe dieses anschaulichen Algorithmusbegriffs kann die Berechenbarkeit von Funktionen und die Entscheidbarkeit sowie die Aufzählbarkeit von Mengen wie folgt definiert werden.

Definition (berechenbare Funktion)
Eine **Funktion** $f : N^k \rightarrow N$ heißt (im anschaulichen Sinne) **berechenbar**, wenn es einen Algorithmus gibt, der zu beliebig vorgegebenem Argument $(x_1, ..., x_k)$ aus dem Definitionsbereich von f nach endlich vielen Schritten den Funktionswert $f(x_1, ..., x_k)$ liefert.

Beispiele:
$f(x) = x+1$
$f(x,y) = x+y$
$f(x,y) = ggT(x,y)$
$f(x_1, x_2, ... , x_k) = (x_{i1}, x_{i2}, ..., x_{ik})$, wobei $i_1, i_2, ..., i_k$ paarweise verschieden sind und $x_{i1} \leq x_{i2} \leq ... \leq x_{ik}$ (Sortieren von k Zahlen)

Es liegt die Vermutung nahe, daß jede arithmetische Funktion, die man präzise definieren kann, auch berechenbar ist. Überraschenderweise gilt das aber nicht. Mehr noch sagt der folgende Satz aus:

Satz: Es gibt überabzählbar viele arithmetische Funktionen, von denen aber nur abzählbar viele berechenbar sind.

Anmerkung: Eine Menge $M \subseteq N$ heißt **abzählbar**, wenn es eine eineindeutige Abbildung von M auf ù oder eine Teilmenge von N gibt. Andernfalls heißt M **überabzählbar**.

Dann gelten folgende Aussagen:
(a) Jede endliche Menge ist abzählbar.
(b) Ist A ein Alphabet (d. h. eine endliche Menge), dann ist die Menge A* aller Wörter über A abzählbar, weil durch die lexikographische Ordnung in natürlicher Weise eine Reihenfolge in A* festgelegt ist.

Beweis des Satzes: Ich beweise zunächst die zweite Aussage des Satzes.
Jeder Algorithmus repräsentiert eine berechenbare Funktion. Folglich gibt es höchstens so viele berechenbare Funktionen wie Algorithmen. Jeder Algorithmus ist seinerseits als Wort über einem Alphabet A darstellbar (sogenannte Niederschrift), d. h. ein Wort aus der Menge A*. Da A* abzählbar ist, muß auch die Menge der Algorithmen und somit die Menge der berechenbaren Funktionen abzählbar sein.
Der erste Teil des Satzes wird mit Hilfe des **Diagonalisierungsverfahrens** indirekt bewiesen: Ich nehme dazu das Gegenteil an, d. h. ,daß es nur abzählbar viele arithmetische Funktionen gibt. Diese kann ich durchnumerieren mit f_1, f_2, f_3 usw. und dann folgende unendliche Matrix betrachten:

	1	**2**	**3**	**4**	**5**	**...**
$\mathbf{f_1}$	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	$f_1(5)$	...
$\mathbf{f_2}$	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	$f_2(5)$	...
$\mathbf{f_3}$	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	$f_3(5)$	...
...	...	...	...	...	...	...

Diese Matrix enthält alle Funktionen, und zwar in der ersten Zeile alle Funktionswerte der Funktion f_1, in der zweiten Zeile alle Funktionswerte der Funktion f_2 usw.
Ich konstruiere nun eine Funktion g, indem ich alle Werte in der Diagonalen ändere. Dazu seien i, j $\in$ N zwei beliebige natürliche Zahlen mit $i \neq j$, und ich definiere dann

$$g(k) := \begin{cases} i, \text{ falls } f_k(k) \neq i \\ j, \text{ falls } f_k(k) = i \end{cases} \quad \text{für alle } k = 1, 2, 3, ...$$

Die so konstruierte Funktion g ist dann offensichtlich von allen Funktionen f_i verschieden und kommt somit in der Folge f_1, f_2, f_3, ... nicht vor. Das ist aber ein Widerspruch zur Annahme, daß diese Folge alle Funktionen erhält. Damit ist der Satz vollständig bewiesen.

Definition (entscheidbare Menge)
Eine **Menge** $M \subseteq N$ heißt (im anschaulichen Sinne) **entscheidbar**, wenn es einen Algorithmus gibt, der zu jedem $x \in N$ nach endlich vielen Schritten die Antwort "ja" oder "nein" auf die Frage liefert, ob $x \in M$ ist.

Beispiele:

(a) Jede endliche Menge ist entscheidbar; ein entsprechender Algorithmus muß lediglich endlich viele Fälle durchmustern, um die Antwort *ja* oder *nein* zu geben.

(b) Die Menge M = {n∈ N | n ist eine gerade Zahl} ist entscheidbar; ein entsprechender Algorithmus testet lediglich, ob die Zahl n durch 2 teilbar ist.

(c) Die Menge M := { n∈ N | n ist eine Primzahl} ist entscheidbar.
Es muß zu jeder vorgegebenen Zahl n 0 ù nur getestet werden, ob es ein m < n mit m … 1 gibt, das die Zahl n teilt.

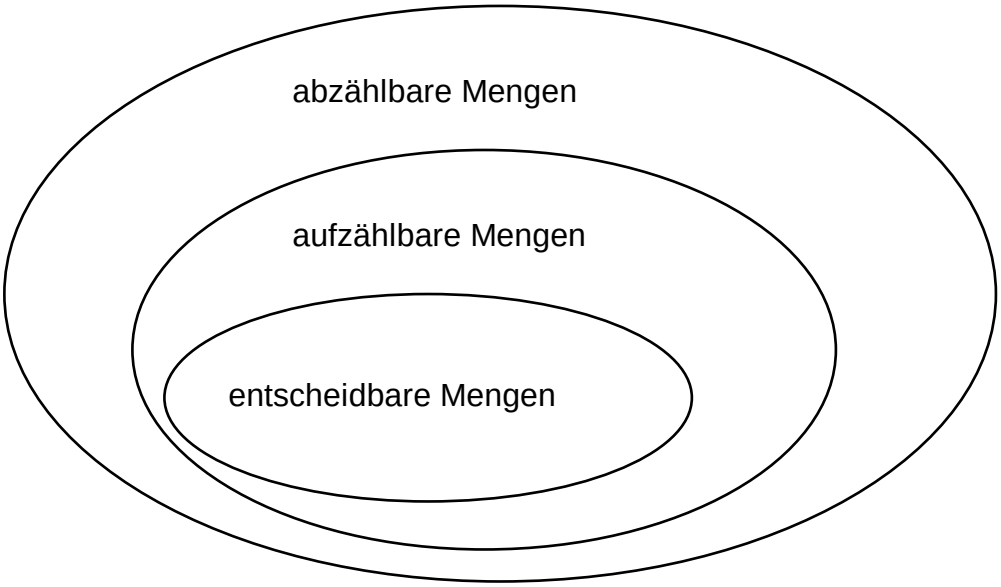
Definition (aufzählbare Menge)
Eine **Menge** M ⊆ N heißt (im anschaulichen Sinne) **aufzählbar**, wenn es eine Funktion f von N auf M gibt (d. h. f(N) = M), die berechenbar ist.

Man sagt dann auch: M wird durch f aufgezählt, d. h. M ={f(0), f(1), f(2),...}
Die verschiedenen Eigenschaften von Mengen hängen nun wie folgt zusammen:

Satz:
(a) M ist aufzählbar ⇒ M ist abzählbar; die Umkehrung gilt i. a. nicht
(b) M ist entscheidbar ⇒ M ist aufzählbar; die Umkehrung gilt i. a. nicht
(c) M ist entscheidbar ⇔ M und N \ M sind aufzählbar

Beispiele:
1. Die Menge M_1 der geraden Zahlen ist aufzählbar
   f(a) := 2a ⇒ f(N) = M_1 = {f(0), f(1), f(2), ...}
2. Die Menge M_2 der ungeraden Zahlen ist aufzählbar
   f(a) := 2a+1 ⇒ f(N) = M_2 = {f(0), f(1), f(2), ...}
3. M_1 und M_2 sind entscheidbar, denn sowohl M als auch N \ M sind aufzählbar

Graphisch kann dieser Sachverhalt wie folgt dargestellt werden:



Speziell von der effektiven Ausführbarkeit der gefundenen Algorithmen - sie ist offensichtlich die eigentliche Schwachstelle der obigen Definitionen - waren ihre Konstrukteure überzeugt, so daß die Notwendigkeit, den Begriff des Algorithmus exakt zu definieren, vorerst gar nicht empfunden wurde. Das änderte sich erst, als zunehmend Probleme auftraten, die sich einer algorithmischen Lösung harnäckig widersetzten und Zweifel an der bis dato dominierenden Auffassung aufkommen ließen, daß jedes mathematische Problem algorithmisch lösbar ist. Beispiele dafür sind:

- *die Vermutung von FERMAT (1601 - 1665)*: Es gibt keine natürliche Zahlen $n \geq 3$, für die positive ganze Zahlen x, y und z mit $x^n + y^n = z^n$ existieren.
- *die Vermutung von GOLDBACH (1690 - 1764)*: Jede gerade Zahl > 4 ist als Summe von zwei Primzahlen darstellbar.
- *das Entscheidungsproblem für den Prädikatenkalkül der 1. Stufe*: Gibt es einen Algorithmus, der für jeden beliebigen booleschen Ausdruck, die Frage beantwortet, ob dieser Ausdruck erfüllbar oder unerfüllbar ist. Dabei heißt ein Ausdruck erfüllbar, wenn es eine Belegung der Variablen des Ausdrucks mit Wahrheitswerten *wahr* oder *falsch* gibt, so daß der Ausdruck als ganzes *wahr* ist
- *das 10 . Problem von HILBERT (1862 - 1943)*: Existiert ein Algorithmus zur Lösung von beliebigen diophantischen Gleichungen?

Die Vermutung von FERMAT wurde erst 1994 bestätigt, während aber die Vermutung von GOLDBACH bis heute weder bewiesen noch widerlegt werden konnte. Für die anderen beiden Probleme konnte bewiesen werden, daß sie nicht entscheidbar sind. (CHURCH, 1936, bzw. MATIJASEVIC, 1970).

2. 2. Präzisierung des Algorithmusbegriffs

Um auf obige Probleme eine negative Antwort der Form "Es gibt nachweislich keinen Algorithmus zur Lösung des Problems!" geben zu können, bedurfte es offensichtlich einer im mathematischen Sinne exakten Definition des Algorithmusbegriffs. Die ersten Präzisierungen wurden Mitte der 30er

Jahre in Arbeiten von HILBERT, GÖDEL, CHURCH, KLEENE, POST und TURING erreicht.

2. 2. 1. Primitiv rekursive Funktionen

Die primitiv-rekursiven Funktionen wurden 1931 von GÖDEL als erste Präzisierung der im anschaulichen Sinne berechenbaren Funktionen eingeführt. Seine grundlegende Idee bestand darin, ausgehend von gewissen Grundfunktionen, deren Berechenbarkeit anschaulich klar war, kompliziertere Funktionen durch Anwendung solcher Operationen zu konstruieren, die auch bei den neuen Funktionen die Berechenbarkeit im anschaulichen Sinne garantieren.

Grundfunktionen:

$N(x) = x + 1$ (Nachfolgerfunktion)

$O(x) = 0$ (Nullfunktion)

$P^n_i(x_1, ..., x_n) = x_i$ (i. Projektion, wobei $1 \leq i \leq n$)

Operationen

- **Einsetzungsprozeß**

  Es seien $h_1, ... h_r$ n-stellige Funktionen und g eine r-stellige Funktion. Gilt nun für beliebige Argumente $(x_1, ... x_n) \in N^n$ die Beziehung

  $$f(x_1, ..., x_n) = g(h_1(x_1, ..., x_n), ..., h_r(x_1, ..., x_n)),$$

  so sagt man, daß **f aus g durch Einsetzung von $h_1, ..., h_r$ entsteht**.

- **Induktionsprozeß**

  Es sei g eine n-stellige Funktion und h eine Funktion mit n+2 Argumenten. Gelten nun für beliebige $(x_1, ..., x_n) \in N^n$ und $y \in N$ die Beziehungen

  (1) $f(x_1, ..., x_n, 0) = g(x_1, ..., x_n)$

  (2) $f(x_1, ..., x_n, y + 1) = h(x_1, ..., x_n, y, f(x_1, ..., x_n, y))$,

  so sagt man, daß **f** durch die beiden Gleichungen (1) und (2) **mit Hilfe von g und h induktiv (oder auch rekursiv) definiert** ist.

Diesen Induktionsprozeß werden wir auch für n=0 anwenden und sagen, daß eine einstellige Funktion f aus einer mit einer Zahl a identischen konstanten, einstelligen Funktion g und einer zweistelligen Funktion h induktiv definiert ist, wenn

$f(0) = a$ und

$f(x+1) = h(x, f(x))$ ist.

Definition (primitiv-rekursive Funktion)

Eine **Funktion** heißt **primitiv-rekursiv**, wenn sie eine der Grundfunktionen ist oder wenn sie aus

diesen Grundfunktionen durch wiederholte Anwendung des Einsetzungs- oder Induktionsprozesses erhalten werden kann.

Beispiele für primitiv-rekursive Funktionen

$f_1(x, y) = x + y$

$f_1(x, 0) = x + 0 = x = P^1_1(x)$

$f_1(x, y + 1) = x + (y + 1) = (x + y) + 1 = f_1(x, y) + 1$

$= h(x , y, f_1(x, y))$

mit $h(x,y,z) = z + 1 = N(P^3_3(x, y, z))$

$f_2(x, y) = x \cdot y$

$f_2(x, 0) = x \cdot 0 = 0 = O(P^2_1(x,y))$

$f_2(x, y + 1) = x \cdot (y + 1) = x \cdot y + x = f_2(x, y) + x$

$= h(x, y, f_2(x, y))$

mit $h(x, y, z) = x + z = f_1(P^3_1(x, y, z), P^3_3(x, y, z))$

$f_3(x, y) = x^y$

$f_3(x, 0) = x^0 = 1 = N(O(P^2_1(x, y)))$

$f_3(x, y + 1) = x^{y+1} = x^y \cdot x = f_3(x, y) \cdot x$

$= h(x, y, f_3(x, y))$

mit $h(x, y, z) = x \cdot z = f_2(P^3_1(x, y, z), P^3_3(x, y, z))$

$f_4(x) = V(x) =$ x -1, falls $x \neq 0$; 0, falls $x = 0$ — Vorgängerfunktion

$V(0) = 0$

$V(x+1) = (x+1) - 1 = x = h(x, f(x))$

mit $h(x, y) = x = P^2_1(x, y)))$

$f_5(x, y) = x -' y =$ x - y, falls $x > y$; 0, sonst — modifizierte Differenz

$f_5(x, 0) = x -' 0 = x = P^1_1(x)$

$f_5(x, y+1) = x -' (y+1) = (x -' y) -' 1 = f_5(x, y) -'1$

$= h(x, y, f_5(x, y))$ mit $h(x, y, z) = z -' 1 = V(P^3_3(x, y, z))$

$f_6(x) = sg(x) =$ 1 für $x \neq 0$; 0 für $x = 0$ — Signum-Funktion

$sg(0) = 0$

$sg(x+1) = 1 = h(x, sg(x))$

$$\text{mit } h(x, y) = 1 = N(O(P^2_1(x, y)))$$

$f_7(x,y) = |x-y|$ Abstandsfunktion

Wir nutzen lediglich aus, daß $|x-y| = (x -'y) + (y-' x)$. Da jeder Summand primitiv rekursiv ist, ist auch die Summe primitiv rekursiv.

$$f_8(x,y,z) = |z \cdot y - x|^{sg(y)} = \begin{cases} |z \cdot y - x| & \text{für } y \neq 0 \\ 1 & \text{für } y = 0 \end{cases}$$ wird später benötigt

Die Funktion f_8 ist offensichtlich genau dann gleich Null, wenn $y \neq 0$ und $z \cdot y = x$ ist. Diese Eigenschaft werden wir später an anderer Stelle ausnutzen.

Anmerkung: Jede primitiv-rekursive Funktion ist total.
Bereits wenige Jahre nach der Entdeckung der primitiv-rekursiven Funktionen gelang der Nachweis, daß mit dem Begriff der primitiv-rekursiven Funktionen aber nicht alle im anschaulichen Sinne berechenbaren Funktionen erfaßt werden, d. h., daß es berechenbare (totale) Funktionen gibt, die nachweislich nicht primitiv-rekursiv sind.
Zum Beweis dieser Aussage benutzt man häufig die sogenannte **ACKERMANN-Funktion**, die durch das folgende (nicht primitive) Rekursionsschema definiert wird:

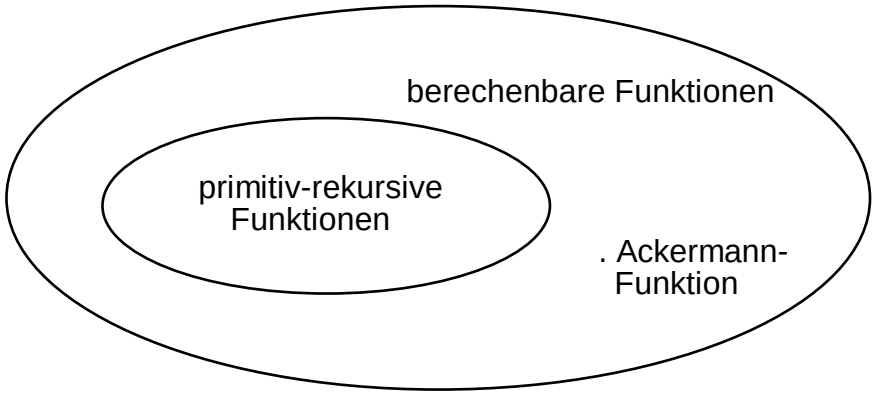
$f(0, y) = y + 1$
$f(x + 1, 0) = f(x, 1)$
$f(x + 1, y + 1) = f(x, f(x + 1, y))$

Ein Pascal-Programm zur Berechnung und Ausgabe der Funktionswerte $f(i, j)$ mit $0 \leq i, j \leq 3$ könnte offenbar wie folgt aussehen:

```
PROGRAM Ackermann_Funktion;
VAR
   i, j : integer;
FUNCTION f(n, m: integer): integer;
BEGIN
  IF n = 0 THEN f := m+1 ELSE IF m = 0 THEN f := f(n - 1, 1) ELSE f := f(n - 1, f(n, m -1))
END;
BEGIN {Hauptprogramm}
  FOR i := 0 TO 3 DO
       FOR j := 0 TO 3 DO
```

```
        writeln('f(', i, ', ', j, ') = ', f(i,j))
END.
```

Folglich ist diese Funktion berechenbar. Der Nachweis, daß die ACKERMANN -Funktion nicht primitiv-rekursiv ist, ist nicht trivial und würde den Rahmen dieser Einführungsvorlesung sprengen. Ich merke lediglich an, daß beim Beweis insbesondere die Tatsache ausgenutzt wird, daß die ACKERMANN-Funktion sehr stark wächst und von keiner primitiv-rekursiven Funktion nach oben beschränkt werden kann. So ist zum Beispiel der Funktionswert f(4, 2) eine Zahl mit über 21000 Ziffern. Interessierte Leser verweise ich auf das Buch von MALCEV (Malcev, 1974).
Damit ergibt sich folgendes Mengendiagramm:



Bei der Suche nach einer die Klasse der primitiv-rekursiven Funktionen umfassenden Klasse berechenbarer Funktionen kam KLEENE zu den partiell-rekursiven Funktionen.

2. 2. 2. Partiell-rekursive Funktionen

Die Erweiterung der Klasse der primitiv rekursiven Funktionen hat KLEENE durch Hinzunahme des sogenannten μ-Operators (auch Minimalisierungsoperator genannt) erreicht.

Definition (μ-Operator):
Es sei g eine k+1-stellige Funktion. Wir sagen, f entsteht durch **Anwendung des μ-Operators** aus g, wenn gilt:

$$f(x_1, ..., x_k) := \mu_y(g(x_1, ..., x_k, y) = 0) = \begin{cases} \min \{y \mid g(x_1, ..., x_k, y) = 0 \text{ und für alle } z < y \\ \qquad \text{ist } g(x_1, ..., x_n, z) \text{ definiert}\} \\ \text{undefiniert sonst} \end{cases}$$

Wir berechnen dazu sukzessive die Werte $g(x_1, ..., x_k, y)$ für y = 0, 1, 2, ... Den kleinsten Wert a, für den wir $g(x_1, ..., x_k, a) = 0$ erhalten, bezeichnen wir mit $\mu_y(g(x_1, ..., x_k, y) = 0)$
Der beschriebene Prozeß zum Auffinden des Wertes $\mu_y(g(x_1, ..., x_k, y) = 0)$ setzt sich in folgenden Fällen aber unendlich fort:
a) der Wert $g(x_1, ..., x_k, 0)$ ist nicht definiert;
b) die Werte $g(x_1, ..., x_k, y)$ sind definiert für y = 0, 1, 2, ..., a-1, aber verschieden von 0 und der Wert $g(x_1, ..., x_k, a)$ ist nicht definiert;

c) die Werte $g(x_1, ..., x_k, y)$ sind für alle y = 0, 1, 2, ... definiert und verschieden von 0.

Beispiele:

1) $f(x) = x^2$

Wir benutzen die Funktion $f_2(x, y) = x \cdot y$ von Seite 7, die primitiv rekursiv ist, und wählen g wie folgt: $g(x,y) = f_2(x, x) - y$. Dann gilt:

$f(x) = \mu_y(f_2(x, x) -' y=0) = \min \{y| x^2 -' y = 0$, und für alle $z < y$ ist $x^2 -' z$ definiert$\}$

Wir berechnen sukzessive $x^2 -' y$ für y = 0, 1, 2 usw. Nach Definition der modifizierten Differenz gilt $x^2 -' y = 0$ für alle $y >= x^2$. Der kleinste Wert y, der diese Bedingung erfüllt, ist offensichtlich x^2.

2) $f(x) = \mu_y(x+1+y= 0)$ ist z. B. die nirgend definierte Funktion, denn es gibt kein y , das x +1 +y = 0 erfüllt.

3) $f(x, y) =$ x/y , falls $y \neq 0$ und y teilt x; undefiniert , sonst

Wir benutzen zum Nachweis der partiellen Rekursivität die Funktion f_8 von Seite 8, die wie folgt definiert war:

$$f_8(x,y,z) = |z \cdot y - x|^{sg(y)} = \begin{cases} |z \cdot y - x| & \text{für } y \neq 0 \\ 1 & \text{für } y = 0 \end{cases}$$

Diese Funktion nimmt offenbar genau dann den Wert 0 an, wenn $y \neq 0$ ist und $z \cdot y = x$. Damit erhalten wir:

$$f(x,y) = \mu_z(f_8(x,y,z)= 0) = \begin{cases} \min \{z| |z \cdot y - x|^{sg(y)} = 0\} \\ \text{undefiniert, sonst} \end{cases}$$

Definition (partiell-rekursive Funktion)

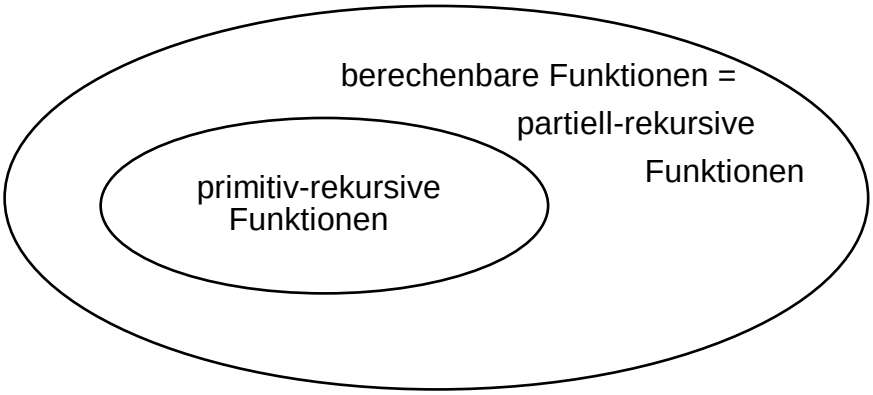
Eine **Funktion** heißt **partiell-rekursiv**, wenn sie eine der Grundfunktionen ist oder wenn sie aus diesen Grundfunktionen durch wiederholte Anwendung des Einsetzungs- oder Induktionsprozesses oder des µ-Operators erhalten werden kann.

Für die Klasse der partiell-rekursiven Funktionen gilt nun die 1936 von Church formulierte Hypothese:

Satz: (Hypothese von Church)

Die Klasse der partiell-rekursiven Funktionen stimmt mit der Klasse der im anschaulichen Sinne berechenbaren Funktionen überein.

Bildlich besteht folgende Beziehung zwischen den verschiedenen Funktionsklassen:



Diese Hypothese kann nie bewiesen werden, da jeder Beweis auf eine neue Hypothese hinausläuft. Ihre Richtigkeit wurde später durch die Äquivalenz aller bekannten Präzisierungen des Berechenbarkeitsbegriffes untermauert.

2. 2. 3. LOOP-, WHILE- und GOTO-Berechenbarkeit von Funktionen

Eine weitere Möglichkeit zur Charakterisierung der primitiv-rekursiven bzw. partiell-rekursiven Funktionen bieten einfache Programmiersprachen mit einerm geeignet eingeschränkten Befehlssatz.

LOOP-Programme

LOOP-Programme sind aus folgenden syntaktischen Komponenten aufgebaut:

Variablen:	x_0 x_1 x_2 ...
Konstanten:	0 1 2 ...
Trennzeichen:	; :=
Operationszeichen:	+ -
Schlüsselwörter:	LOOP DO END

Die ***Syntax*** von LOOP-Programmen kann wie folgt induktiv definiert werden:

- Jede Wertzuweisung der Form

  $\mathbf{x_i := x_j + c}$ bzw. $\mathbf{x_i := x_j - c}$

  ist ein LOOP-Programm (wobei c eine beliebige Konstante ist).
- Falls $\mathbf{P_1}$ und $\mathbf{P_2}$ bereits LOOP-Programme sind, dann auch

  $\mathbf{P_1 ; P_2}$

- Falls **P** ein LOOP-Programm ist und $\mathbf{x_i}$ eine Variable, dann ist auch

  LOOP x_i DO P END

  ein LOOP-Programm.
- Sonst nichts.

Die ***Semantik*** von LOOP-Programmen ist wie folgt definiert:
Bei einem LOOP-Programm, das eine k-stellige Funktion berechnen soll, gehen wir davon aus, daß dies mit den Startwerten $\mathbf{n_1}$, ..., $\mathbf{n_k}$ in den Variablen $\mathbf{x_1}$, ..., $\mathbf{x_k}$ gestartet wird und alle anderen vorkommenden Variablen den Anfangswert 0 haben.
Die Wertzuweisung $\mathbf{x_i := x_j + c}$ wird wie üblich interpretiert: der neue Wert der Variablen x_i berechnet sich zu $\mathbf{x_j + c}$, wobei c eine Konstante ist. Bei $\mathbf{x_i := x_j - c}$ wird die modifizierte Subtraktion verwendet, also falls $c > x_j$, so wird das Resultat auf 0 gesetzt.
Ein LOOP Programm der Form $\mathbf{P_1 ; P_2}$ wird so interpretiert, daß zuerst $\mathbf{P_1}$ und dann $\mathbf{P_2}$ auszuführen ist.
Ein LOOP-Programm der Form **LOOP x_i DO P END** wird so interpretiert, daß das Programm **P** sooft ausgeführt wird, wie der Wert der Variablen $\mathbf{x_i}$ zu Beginn angibt. D. h. ein Ändern des Wertes von $\mathbf{x_i}$ im Innnern von **P** hat keinen Einfluß auf die Anzahl der Wiederholungen.
Das Resultat der Berechnung eines LOOP-Programms ergibt sich als Wert der Variablen x_0.

Definition (LOOP-berechenbar)
Eine k-stellige Funktion **f** heißt **LOOP-berechenbar**, wenn es ein LOOP-Programm P gibt, das f in dem Sinne berechnet, daß P, gestartet mit den Werten $\mathbf{n_1}$, ..., $\mathbf{n_k}$ in den Variablen $\mathbf{x_1}$, ..., $\mathbf{x_k}$ (und 0 in den restlichen Variablen) stoppt mit dem Wert $\mathbf{f(n_1, ..., n_k)}$ in der Variablen $\mathbf{x_0}$.

Es ist sofort klar, daß alle LOOP-berechenbaren Funktionen totale, also überall definierte Funktionen sind, denn jedes LOOP-Programm stoppt zwangsläufig nach endlicher Zeit. Mit LOOP-Programmen können nämlich keine unendlichen Schleifen programmiert werden.

Beispiele:
1) Addition $x_0 = x_1 + x_2$

$x_0 := x_1$;
LOOP x_2 DO $x_0 := x_0 + 1$ END

2) Simulation der einseitigen Alternative IF x = 0 THEN P_1 END

y := 1;
LOOP x DO y := 0 END;

LOOP y DO P_1 END;

3) Simulation der einseitigen Alternative IF $x \neq 0$ THEN P_1 END

y := 0;
LOOP x DO y := 1 END;
LOOP y DO P_1 END;

4) Multiplikation $x_0 = x_1 \cdot x_2$

x_0 = 0;
LOOP x_2 DO LOOP x_1 DO $x_0 := x_0 + 1$ END

5) Simulation der zweiseitigen Alternative IF x = 0 THEN P_1 ELSE P_2 END

IF x = 0 THEN P_1;
IF $x \neq 0$ THEN P_2

Satz: (ohne Beweis)
Die Klasse der primitiv-rekursiven Funktionen stimmt mit der Klasse der LOOP-berechenbaren Funktionen überein.

WHILE-Programme

Erweitern wir die LOOP-Programme noch um die sogenannte WHILE-Schleife kommen wir zur Klasse der WHILE-Programme. Die ***Syntax*** von WHILE-Programmen enthält alle Konzepte, wie sie bei LOOP-Programmen vorkommen, mit folgendem Zusatz:

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

  WHILE $x_i \neq 0$ DO P END

  ein WHILE-Programm.

Die ***Semantik*** dieses neuen Konstrukts ist so definiert, daß das Programm **P** solange wiederholt ausgeführt wird, wie der Wert der Variablen $\mathbf{x_i}$ ungleich Null ist.

Beispiele:

1) Simulation der LOOP-Schleife durch eine WHILE-Schleife

LOOP x DO P END

y := x {y komme in P nicht vor!!}
WHILE y … 0 DO y := y -1; P END

Folglich können wir künftig auf LOOP-Schleifen verzichten.

2) Simulation einer Endlosschleife: Sei $x \neq 0$ und das Programm P ändere den Wert von y nicht.

y := x;
WHILE $y \neq 0$ DO P END

Definition (WHILE-berechenbar)
Eine k-stellige Funktion **f** heißt **WHILE-berechenbar**, wenn es ein WHILE-Programm P gibt, das f in dem Sinne berechnet, daß P, gestartet mit den Werten $\mathbf{n_1}$, ..., $\mathbf{n_k}$ in den Variablen $\mathbf{x_1}$, ..., $\mathbf{x_k}$ (und 0 in den restlichen Variablen) stoppt mit dem Wert $\mathbf{f(n_1, ..., n_k)}$ in der Variablen $\mathbf{x_0}$, sofern $\mathbf{f(n_1, ..., n_k)}$ definiert ist. Ansonsten stoppt P nicht.

Wiederum ohne Beweis können wir dann folgenden Satz formulieren:

Satz:
Die Klasse der partiell-rekursiven Funktionen stimmt mit der Klasse der WHILE-berechenbaren Funktionen überein.

GOTO-Programme
GOTO-Programme bestehen aus Sequenzen von Anweisungen A_i, die jeweils durch eine Marke M_i eingeleitet werden: $M_1: A_1$; $M_2: A_2$; ...; $M_k: A_k$

Als Anweisungen A_i sind zugelassen:

Wertzuweisungen:	$x_i := x_j + c$ bzw. $x_i := x_j - c$
unbedingter Sprung:	**GOTO** M_i
bedingter Sprung:	**IF** $x_i = c$ **THEN GOTO** M_j
Stopanweisung:	HALT

Beim Niederschreiben von GOTO-Programmen lassen wir Marken, die niemals angesprungen werden, in der Regel weg.
Die Semantik solcher Programme sollte klar sein. GOTO-Berechenbarkeit definiert man analog der Definition von WHILE-Berechenbarkeit. Es ist klar, daß GOTO-Programme auch in unendliche Schleifen geraten können (M_1: **GOTO** M_1).

Satz:
Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden und umgekehrt.

Beweis:
1)

geg.: WHILE $x_i \neq 0$ DO P END

Lösung:
M_1: IF $x_i = 0$ THEN GOTO M_2;
P;
GOTO M_1
M_2: ...

2)

geg.: M_1: A_1; M_2: A_2; ..., M_k: A_k

Lösung:
c := 1;
WHILE c $\neq 0$ DO
IF c = 1 THEN A_1' END;
IF c = 2 THEN A_2' END;
...
IF c = k THEN A_k' END;
END

Dabei ist A_i' wie folgt definiert:

A_i' =
$x_j := x_l$ " a ; c := c + 1 , falls $A_i = x_j := x_l$ " a
c := n, falls A_i = GOTO M_n
IF x_j = a THEN c := n
ELSE c := c + 1 END, falls A_i = IF x_j = a THEN GOTO M_n
c := 0, falls A_i = HALT

Zusammenfassend gilt:

Satz:
Es sei f eine beliebige k-stellige zahlentheoretische Funktion. Dann gilt
f partiell rekursiv $\Leftrightarrow$ f WHILE-berechenbar $\Leftrightarrow$ f GOTO-berechenbar

Aus der Sicht der modernen Rechentechnik verdienen insbesondere die Definitionen erhöhte Aufmerksamkeit, bei denen das Wesen des Algorithmus durch theoretische Maschinenmodelle offenbart wird. Das erste Maschinenmodell lieferte 1936 der englische Mathematiker A.M.Turing mit der nach ihm benannten **Turingmaschine** (siehe Kapitel 3.2.4).
Ein weiteres Modell entwickelten 1963 die englischen Mathematiker J. C. Sheperdson und H. E. Sturgis, die sogenannte **Registermaschine**. Sie hat gegenüber der Turingmaschine den Vorteil, daß sie sehr stark real existierenden Computern entspricht , und ist deshalb für die Schule besonders gut geeignet.

2. 2. 4. Registermaschinen-Berechenbarkeit

Zunächst einige vorbereitetende Definitionen

Definitionen

- Alphabet V $=_{def.}$ endliche, nichtleere Menge von unterscheidbaren Grundzeichen (oder Buchstaben), so daß jede Folge von Zeichen auf genau eine Art mittels Grundzeichen aus V darstellbar ist.

  V = { | , || } und V = { 1, 4, 41 } sind z. B. keine Alphabete
- w Wort über V $=_{def}$ w endliche Folge von Grundzeichen
- leeres Wort 7 $=_{def}$ enthält kein Grundzeichen
- V^* $=_{def}$ Menge aller Wörter über V
-

Nun sei V ein Arbeitsalphabet, d. h. eine beliebige nichtleere, endliche Menge. Eine Registermaschine (RM) über V besteht dann aus potentiell unendlich vielen Registern R_0, R_1,...,R_m , die Wörter aus V* enthalten können, und einem Programm als endliche Folge von Befehlen der folgenden Art:

Syntax	**Semantik**
A(v;k)	Addiere das Zeichen 'v' zur Zeichenkette im k. Register.
L(k) :	Lösche das 1.Zeichen der Zeichenkette im k. Register.
S(v;k)<p>	Springe zum Befehl mit der Marke p, wenn die Zeichenkette im k.Register mit dem Zeichen 'v' beginnt. Ansonsten arbeite den nächsten Befehl des Programms ab.
S(k)<p>	Springe zum Befehl mit der Marke p, wenn die Zeichenkette im k. Register leer ist.
S<p>	Springe zum Befehl mit der Marke p.
E	Stopanweisung

Definition (**RM-berechenbar**)
Eine k-stellige Funktion **f** heißt Registermaschinen-berechenbar (**RM-berechenbar**), wenn es ein Registermaschinen-Programm gibt, das folgendes leistet:
Zu Beginn stehe im Register R_i das Argument x_i (i=1,2,...,k) der Funktion f und die anderen Register seien leer. Die Registermaschine stoppt dann und nur dann in der Situation, daß im Register R_0 der Funktionswert $f(x_1, ..., x_k)$ steht und alle anderen Register leer sind, wenn das k-Tupel $(x_1, ..., x_k)$ im Definitionsbereich der Funktion f liegt.

Beispiel:

1:S(1)<11>, 2:S(a;1)<5>, 3:S(b;1)<8>, 4:S<1>, 5:A(a;0), 6:L(1), 7:S<1>, 8:A(b;0), 9:L(1), 10:S<1>, 11:S(2)<21>, 12:S(a;2)<15>, 13:S(b;2)<18>, 14:S<11>, 15:A(a;0), 16:L(2), 17:S<11>,

18:A(b;0), 19:L(1), 20:S<11>, 21:E

Dieses Programm berechnet offensichtlich die 2-stellige Funktion f(w, v) = w + v , die für alle Zeichenketten w und v über dem Alphabet {a, b}definiert ist. Bei Einschränkung auf das Alphabet V = { | }(sogenannte unäre Notation oder Bierdeckel-Notation) handelt es sich offensichtlich um die Berechnung der arithmetischen Funktion f(x, y) = x + y.

Übungsaufgaben:
1) arithmetische Funktionen in unärer Codierung (z.B. f(x,y) = x*y, $f(x, y) = x^y$)
2) Wortumkehrfunktion (z. B. f(aaabb) = bbaaa)
3) Umwandlung von (unär dargestellten) Dezimalzahlen in Dualzahlen (Divisionsalgorithmus)

Satz :
Die Klasse der RM-berechenbaren Funktionen stimmt mit der Klasse der im anschaulichen Sinne berechenbaren Funktionen überein.

Dieser Satz ist eine spezielle Form der von CHURCH formulierten Hypothese (**Churchsche Hypothese**). Ihre Richtigkeit wird dadurch gestützt, daß die Übereinstimmung zwischen der hier definierten RM-Berechenbarkeit von Funktionen und allen anderen bisher bekannten Präzisierungen des Berechenbarkeitsbegriffs bewiesen werden konnte. So ist z. B. der folgende Satz ganz offensichtlich:

Satz:
Es sei f eine beliebige k-stellige zahlentheoretische Funktion. Dann gilt
f GOTO-berechenbar $\Leftrightarrow$ f RM berechenbar

2. 3. Komplexität von Algorithmen und Problemen

Unter der **Komplexität** eines Algorithmus bzw. eines Problems versteht man den Aufwand an Betriebsmitteln wie z. B. Rechenzeit (Zeitkomplexität) oder Speicherplätze (Speicherkomplexität), die zur Abarbeitung des Algorithmus bzw. zur Lösung des Problems erforderlich sind. Aufgabe der Komplexitätstheorie ist es, einerseits für konkrete Probleme möglichst effiziente Lösungsalgorithmen anzugeben und andererseits untere und obere Schranke für die Komplexität des Problems zu bestimmen.
Die Zeitkomplexität eines Algorithmus wurde 1960 von Rabin eingeführt. Sie ist eine arithmetische Funktion, die den Aufwand an Rechenoperationen in Abhängigkeit vom Umfang des Problems (Anzahl der Eingabedaten, Ordnung der Matrix, Grad des Polynoms o.ä.) ausdrückt. Man

unterscheidet dabei zwischen dem Aufwand im Mittel (**average case**) und dem Aufwand im schlechtesten Fall (**worst case**). Interessiert man sich für die Laufzeit eines Algorithmus zur Lösung eines Problems vom Umfang n im Mittel bzw. im schlechtesten Fall, dann betrachtet man die Laufzeit für sämtliche Beispiele des Problems vom Umfang n und bestimmt davon den Mittelwert bzw. das Maximum. Dabei wird nicht unterschieden, ob es sich um eine Anweisung oder einen Test handelt. Natürlich wäre es wünschenswert, das Verhalten eines Algorithmus vollständig zu kennen und somit die Funktion des Rechenzeitaufwandes in Abhängigkeit vom Umfang des Problems exakt aufschreiben zu können, aber in der Praxis kann diese Aufgabe nur mit großen Schwierigkeiten gelöst werden. Oft muß man sich damit begnügen, den Zeitaufwand größenordnungsmäßig abzuschätzen und nur das asymptotische Algorithmusverhalten zu kennen. Um solche Größenordnungen von Funktionen auszudrücken, hat sich die sogenannte *Groß-Oh*-Notation von LANDAU bewährt:

Definition: (Zeitkomplexität)
Ein Algorithmus mit einem Rechenzeitaufwand A(n) hat die Zeitkomplexität O(g(n)), sofern es eine Konstante $c > 0$ gibt, so daß $A(n) \leq c \cdot g(n)$ für alle $n \in N$ gilt.
Kann die Zeitkomplexität eines Algorithmus nach oben mit einem geeigneten $k > 0$ durch $g(n) = n^k$ abgeschätzt werden, dann nennt man den Algorithmus polynomial. Ansonsten spricht man von exponentiellen Algorithmen..

Praktisch bedeutet das: Wenn der Rechenzeitaufwand eines Algorithmus nach oben z. B. durch ein Polynom $a_k n^k + a_{k-1} n^{k-1} + \ldots a_1 n + a_0$ mit $a_k \neq 0$ abgeschätzt werden kann, dann hat der zugehörige Algorithmus die Zeitkomplexität $O(n^k)$. Für die Abschätzung des Rechenzeitaufwandes ist also ledig-lich von Interesse, daß die Gesamtzahl der benötigten Rechenoperationen ein Vielfaches von n^k nicht übersteigt. Auf konstante Faktoren und kleinere Potenzen kommt es dabei nicht an. Obwohl n log n kein Polynom ist, ist ein Algorithmus mit der Komplexität O(n log n) auch als polynomial anzusehen, denn $n \log n < n^2$.

Beispiel:
Sortieralgorithmus "Ripplesort"
Gegeben sei eine Folge $(a_i)_{i=1,\ldots,n}$ von Zahlen, die in aufsteigender Reihenfolge angeordnet werden sollen. Gehen Sie beim Sortieren wie folgt vor:

- Vergleichen Sie das erste Glied der Zahlenfolge nacheinander mit allen nachfolgenden Gliedern: Immer wenn eines der nachfolgenden Glieder kleiner ist als das jeweils erste, dann vertauschen Sie dieses Glied mit dem ersten Glied.
- Ist das erste Glied der Zahlenfolge mit allen Gliedern der Zahlenfolge verglichen worden, dann bleibt es im nächsten Durchlauf unberücksichtigt und das Verfahren beginnt von vorn

mit dem zweiten Glied.

- Das Verfahren wird so bis zum vorletzten Glied durchgeführt.

Beispiel 1:

$\mathbf{a}_1$ =	4	4	1	1	1	1	1

$\mathbf{a}_2$ =	5	5	5	5	4	2	2

$\mathbf{a}_3$ =	1	1	4	4	5	5	4
$\mathbf{a}_4$ =	2	2	2	2	2	4	5

1. Durchlauf	2. Durchlauf	3. Durchlauf
(3 Vergleiche)	(2 Vergleiche)	(1 Vergleich)

Bei n Zahlen sind im 1. Durchlauf n-1 Vergleiche, im 2. Durchlauf n-2 Vergleiche usw. im letzten Durchlauf ein Vergleich durchzuführen. Der worst case liegt genau dann vor, wenn die Zahlen in umgekehrter Reihenfolge vorliegen, d.h. a[i] := n+1-i für i =1, 2, 3,... n. In diesem Fall ist nach jedem Vergleich ein Tausch durchzuführen. Für jeden Tausch werden jeweils 3 Wertzuweisungen (Dreieckstausch) benötigt. Insgesamt sind also

$$A(n) = 3\,[(n-1) + (n-2) + \ldots + 1] = 3n^2/2 - 3n/2$$

Operationen bei n Zahlen erforderlich. Folglich hat Ripplesort die Zeitkomplexität $O(n^2)$, denn

$$A(n) \leq 2 \cdot n^2$$

Die quadratische Abhängigkeit der Anzahl der Rechenoperationen vom Umfang der Eingabedaten ist auch sehr gut in einem Computerexperiment zu erkennen, das im Unterricht nicht fehlen sollte. Die Leistungsgrenze von Ripplesort wird bei folgender Überlegung deutlich:
Wieviel Zeit benötigt ein fiktiver Computer, der 10^6 Operationen pro Sekunde ausführen kann, zum Sortieren von 60 Millionen Daten mit Hilfe von Ripplesort?

$$60 \cdot 10^6 \cdot 60 \cdot 10^6/10^6 = 3600 \cdot 10^6 \text{ Sekunden} = 114 \text{ Jahre.}$$

Daraus ergibt sich die Notwendigkeit der Suche nach effizienteren Lösungsalgorithmen wie **Quick-sort** u. a.

Quicksort - ein schneller Sortieralgorithmus

- Nehmen Sie das mittlere Glied (arithmetisches Mittel der Indizes) der gegebenen Zahlenfolge als sogenanntes "Trennelement" (T) und schreiben Sie alle anderen Glieder unter Beibehaltung der Reihenfolge davor (wenn sie kleiner oder gleich sind) oder dahinter (wenn sie größer sind).

- Mit den so entstandenen Teilfolgen, die durch das Trennelement getrennt werden, verfahren Sie ebenso.
- Das Verfahren wird abgebrochen, wenn alle Teilfolgen höchstens ein Glied enthalten.

Beispiel:

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}
8	5	12	4	10	4	7	15	6	11	16	9	7	6
5	4	4	6	7	6	7	8	12	10	15	11	16	9
						T							
4	4	5	6	7	6	7	8	12	10	11	9	15	16
	T					**T**						**T**	
4	4	5	6	6	7	7	8	9	10	12	11	15	16
	T			**T**		**T**			**T**			**T**	
4	4	5	6	6	7	7	8	9	10	11	12	15	16
	T	**T**		**T**		**T**	**T**		**T**		**T**	**T**	

Quicksort hat im **worst case** wie Ripplesort die Zeitkomplexität $O(n^2)$, aber im **average case** besitzt Quicksort die Zeitkomplexität $O(n \cdot \log n)$ und benötigt dann für das Sortieren von 60 Millionen Zahlen nur etwa 25 Minuten.

Definition (Klasse der in polynomialer Zeit lösbaren Probleme)
Ein Problem wird in polynomialer Zeit gelöst, wenn es einen polynomialen Algorithmus zur Lösung des Problems gibt.
P bezeichne die Klasse aller Probleme, die in polynomialer Zeit gelöst werden können

In der Praxis hat sich gezeigt, daß es für die meisten polynomialen Probleme Algorithmen gibt, deren Rechenzeitaufwand von der Größenordnung $O(n^3)$ oder kleiner ist.
Gegenwärtig ist man übereinstimmend der Meinung, daß höchstens polynomiale Algorithmen **praktische Bedeutung** haben. Algorithmen mit exponentiellem Rechenzeitaufwand sind für die praktische Anwendung nur bedingt geeignet, da die benötigte Rechenzeit schon bei relativ kleinen Problemumfängen extrem hoch ist, wovon man sich durch einen Blick auf den Werteverlauf einfacher Exponentialfunktionen sofort überzeugen kann:

n	**10**	**100**	**1000**
$\mathbf{n^3}$	10^3	10^6	10^9
$\mathbf{2^n}$	1024	$1{,}27 \cdot 10^{30}$	$1{,}05 \cdot 10^{301}$

Wir wollen uns nun für die Klasse der Probleme interessieren, für die beim gegenwärtigen Stand

der Mathematik kein polynomialer Lösungsalgorithmus bekannt ist und vermutlich auch nicht existiert. Derartige Probleme sind praktisch unlösbar, da die benötigte Rechenzeit schon bei relativ kleinen Problemumfängen extrem hoch ist. Unglücklicherweise ist diese Klasse sehr umfangreich und täglich kommen neue Probleme hinzu.

Das Rundreiseproblem
Es seien k und n natürliche Zahlen und D= (d_{ij}) eine Matrix vom Typ (n,n). d_{ij} sei dabei die Entfernung von der Stadt **i** zur Stadt **j**. Ein Handelsreisender soll, ausgehend von der 1.Stadt, n-1 andere Städte genau einmal durchreisen und am Ende wieder in die Ausgangsstadt zurückkehren. Gibt es eine Rundreise derart, daß die Summe der zurückgelegten Entfernungen kleiner oder gleich k ist?

Das Problem ist offensichtlich entscheidbar, denn es gibt nur endlich viele verschiedene Rundreisen. Ein trivialer Lösungsalgorithmus könnte darin bestehen, daß alle (n-1)! Rundreisen durchgemustert werden und unter allen die eine kürzeste ausgewählt wird. Dieser Algorithmus hätte die Zeitkomplexität O(n!).

Bei Verwendung eines fiktiven Computers, der wiederum 10^6 Operationen pro Sekunde ausführen kann, ergeben sich folgende Werte:

Anzahl n	16	18	20
O(n!)	242Tage	203 Jahre	771 Jahrhunderte

Rucksackproblem
Es seien k und n natürliche Zahlen und m = $(m_1, ..., m_n)$, und w = $(w_1, ..., w_n)$ Vektoren der Länge n. Dabei sei m_i die Masse des Gegenstandes G_i, w_i der Wert des Gegenstandes G_i und T die Tragfähigkeit eines Rucksacks. Gibt es eine Auswahl der Gegenstände derart, daß der Rucksack nicht platzt und der Gesamtwert größer oder gleich k ist

Ein exakter Lösungsalgorithmus könnte auch hier darin bestehen, alle Auswahlmöglichkeiten durchzumustern und unter denen, deren Gesamtmasse die Tragfähigkeit des Rucksackes nicht übersteigt, eine Auswahl mit maximalem Gesamtwert zu ermitteln. Dieser Algorithmus wäre exponentiell und hätte die Zeitkomplexität $O(2^n)$.

Bei Verwendung eines fiktiven Computers, der wiederum 10^6 Operationen pro Sekunde ausführen kann, ergeben sich folgende Werte:

Anzahl n	10	20	30	40	50
$O(2^n)$.	0,001 Sekunden	1,0 Sekunden	17,9 Minuten	12,7 Tage	35,7 Jahre

Die Beispiele machen deutlich, daß exponentielle Algorithmen praktisch keine Bedeutung haben, da ihr Rechenzeitaufwand schon bei kleinen Problemumfängen "astronomisch" ist.
Sowohl für das Rundreiseproblem als auch für das Rucksackproblem sind bis heute aber keine polynomialen Algorithmen bekannt . Sie existieren vermutlich auch nicht!!

Nichtdeterministische Algorithmen
Um die Klasse der Probleme, für die beim gegenwärtigen Stand der Mathematik kein polynomialer Lösungsalgorithmus bekannt, zu charakterisieren, ist es erforderlich, den Begriff des nichtdeterministischen Algorithmus einzuführen. Er ist eine Erweiterung des intuitiven Algorithmusbegriffs in dem Sinne, daß auf die Forderung verzichtet wird, daß der nächstfolgende Schritt bei der Abarbeitung eines Algorithmus stets eindeutig bestimmt ist. Man nimmt statt dessen an, daß man den nächsten Schritt zu jedem Zeitpunkt aus einer endlichen Menge von möglichen Schritten auswählen (raten) kann .

Definition (nichtdeterministischer Algorithmus
Der Begriff des nichtdeterministischen Algorithmus ist eine Erweiterung des anschaulichen Algorithmusbegriffs in dem Sinne, daß auf die Forderung verzichtet wird, daß der nächstfolgende Schritt bei der Abarbeitung eines Algorithmus stets eindeutig bestimmt ist. Man nimmt statt dessen an, daß man den nächsten Schritt zu jedem Zeitpunkt aus einer endlichen Menge von möglichen Schritten auswählen
Die Abarbeitung eines nichtdeterministischen Algorithmus erfolgt in zwei Phasen. In der ersten Phase, der **Ratephase**, wird ein Lösungskandidat geraten und in der zweiten Phase wird getestet, ob der Lösungskandidat tatsächlich eine Lösung darstellt.

Die Nichtdeterminiertheit liegt also lediglich in der ersten Phase, während die zweite Phase völlig deterministisch verläuft.
Betrachten wir dazu das Rundreiseproblem:
Um eine Lösung zu bestimmen, können Sie wie folgt vorgehen: Sie raten zunächst nichtdeterministisch irgendeine Rundreise. Anschließend benutzen Sie einen polynomialen Algorithmus zur Bestimmung der Summe der Teilstrecken dieser Rundreise und zum Test, ob diese Summe kleiner oder gleich k ist.
Analog können Sie beim Rucksackproblem zunächst eine Auswahl an Gegenständen nichtdeterministisch raten und anschließend mit einem deterministischen Algorithmus zur Bestimmung der

Gesamtmasse und des Gesamtwertes der eingepackten Gegenstände sowie zum Test, ob dieser Gesamtwert größer oder gleich k ist.

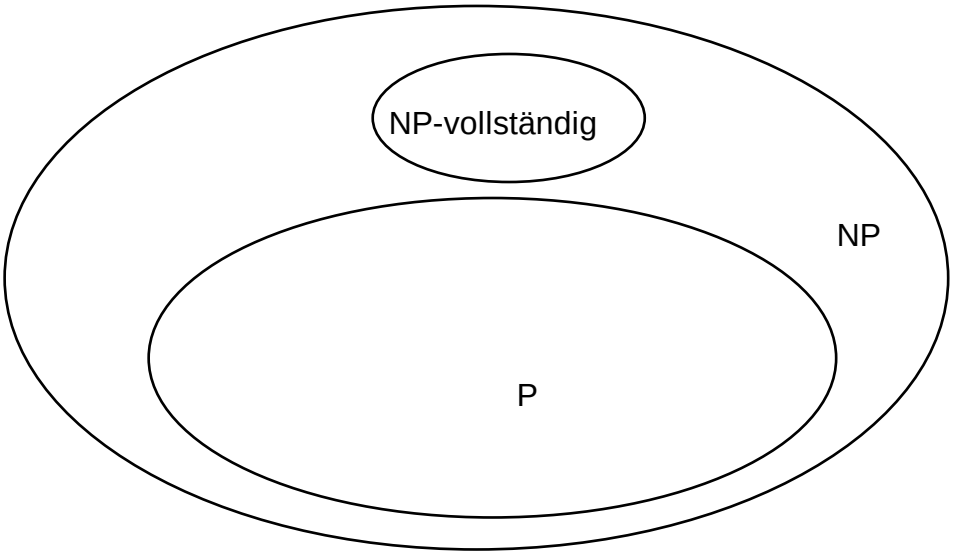
Definition (Klasse NP)
NP bezeichne die Klasse aller Probleme, die mit nichtdeterministischen Algorithmen im oben beschriebenen Sinne in polynomialer Zeit gelöst werden können.

Dann gilt offensichtlich die Aussage

$$\mathbf{P} \subseteq \mathbf{NP},$$

denn jeder deterministische Algorithmus ist ein Spezialfall eines nichtdeterministischen. Ein noch immer offenes Problem der Informatik ist die Frage, ob diese Inklusion echt ist, d. h. , ob $\mathbf{P} \subset \mathbf{NP}$ gilt. Das ist der Inhalt des sogenannten **P**-**NP**-Problems der Informatik.
Da gegenwärtig weltweit vermutet wird, daß **P** eine echte Teilmenge von **NP** ist, wird nach Problemen gesucht, die in **NP**, aber nicht in **P** liegen. Diese Suche führte zur Klasse der sogenannten **NP-vollständigen** Probleme:

Definition (NP-vollständiges Problem)
Ein Problem heißt **NP-vollständig**, wenn es erstens zu Klasse **NP** gehört und zweitens vollständig in folgendem Sinne ist: Findet man einen deterministischen polynomialen Algorithmus für das Problem, so kann man aus diesem einen polynomialen Algorithmus für jedes Problem aus **NP** ableiten und dann wäre **P**=**NP**.



Die **NP**-vollständigen Probleme sind sozusagen die "schwierigsten" Probleme. Wenn $\mathbf{P} \neq \mathbf{NP}$ ist, dann kann man kein **NP**-vollständiges Problem mit einem deterministischen Algorithmus in polynomialer Zeit lösen.

Eine Lösung des **P**-**NP**-Problems hätte weitreichende Konsequenzen: Wäre nämlich **P** = **NP**, dann würde mit Sicherheit auch für die Probleme ein polynomialer Lösungsalgorithmus existieren, für die bis heute nur exponentielle bekannt sind.
Sowohl das Rundreiseproblem als auch das Rucksackproblem gehören nachweislich zur Klasse der **NP**-vollständigen Probleme. Die exakten Beweise dieser Aussage sind sehr umfangreich und können im Rahmen dieser Arbeit nicht nachvollzogen werden. Interessierten Lehrerinnnen und Lehrern empfehle ich das Buch von M. R. Garey und D. S. Johnson, das als Standardwerk für die Theorie der NP-Vollständigkeit gilt.

Algorithmisch unlösbare Problem
Bereits auf S. 4 sind Probleme genannt, die nachweislich nicht entscheidbar sind. Ein weiteres Beispiel ist das sogenannte Halteproblem, das wie folgt formuliert werden kann:

Halteproblem
Gibt es ein PASCAL-Programm, mit dessen Hilfe man für jedes beliebige PASCAL-Programm entscheiden kann, ob es mit jeder beliebigen Eingabe nach endlich vielen Schritten abbricht oder nicht?

Wir wollen im folgenden exakt beweisen, daß es ein solches PASCAL-Programm nicht geben kann.
Vorbemerkungen:

- Jedes PASCAL-Programm kann als Zeichenkette dargestellt werden und als solches dann auch als Eingabe in PASCAL-Programmen verwendet werden.
- PASCAL-Programme sind folglich auf PASCAL-Programme anwendbar, und speziell kann jedes PASCAL-Programm auf sich selbst angewendet werden.

Anstelle des obigen Halteproblems ist dann zunächst folgende Problem zu formulieren:

Selbstanwendbarkeitsproblem
Gibt es ein PASCAL-Programm, das von jedem beliebigen PASCAL-Programm, das auf sich selbst angewendet wird, entscheidet, ob es nach endlich vielen Schritten stoppt oder nicht (
Dieses Problem ist offensichtlich ein Spezialfall des obigen Halteproblems, und folglich gilt, daß aus seiner algorithmischen Unlösbarkeit die Unlösbarkeit des Halteproblems folgen würde.

Nach diesen Vorbetrachtungen kann der exakte Beweis als indirekter Beweis wie folgt geführt werden:

Beweis:
Verabredung: Wir nennen ein Programm **selbststoppend**, wenn es bei Anwendung auf sich selbst

nach endlich vielen Schritten stoppt.
Angenommen, es gäbe ein PASCAL-Programm, das entscheidet, ob ein PASCAL-Programm mit seinem eigenen Programmtext als Eingabe nach endlich vielen Schritten stoppt oder nicht. Das Programm könnte z.B. "Stoptest" heißen. Es benutzt eine logische Variable "stop", die im Laufe der Abarbeitung dann und nur dann den Wert TRUE erhält, wenn das Programm als selbststoppend erkannt wurde. Der Einfachheit halber kann ferner o.B.d.A. angenommen werden, daß das Programm "Stoptest" ganz am Ende die Anweisung zur Ausgabe des positiven bzw. negativen Ergebnisses des Stoptests enthält:

```
PROGRAM Selbststop;
...
VAR
...
stop : BOOLEAN;
...
BEGIN
  ...
  IF stop THEN writeln ('Das Programm ist selbststoppend.') ELSE
                writeln (`Das Programm ist nicht selbststoppend. ')
END.
```

Wenn ein solches Programm "Stoptest" existiert, dann existiert auch das folgende Programm, das sinnvollerweiser "Seltsam" heißen könnte:

```
PROGRAM Seltsam;
...
VAR
...
stop : BOOLEAN;
...
BEGIN
  ...
  IF stop THEN
            BEGIN
               writeln ('Das Programm ist selbststoppend.');
               WHILE true DO ;
             END
          ELSE writeln (`Das Programm ist nicht selbststoppend. ')
```

END.

Der Unterschied zwischen den beiden Programmen besteht offensichtlich nur darin, daß hinter die positive Ausgabe eine Endlosschleife eingebaut wurde.
Sodann kann auch für dieses Programm "Seltsam" die Frage aufgeworfen werden, ob es selbststoppend ist oder nicht.
Angenommen, das Programm "Seltsam" ist selbststoppend. Startet man es dann mit seinem eigenen Programm als Eingabe, so wird zunächst das Programm "Stoptest" abgearbeitet und die boolesche Variable erhält den Wert TRUE. Das bedingt, daß die positive Antwort " Das Programm ist selbststoppend." ausgegeben wird, der Computer anschließend aber in die Endlosschleife gerät und im Widerspruch zur Annahme kein Stop erfolgt.
Angenommen, daß Programm "Seltsam" ist nicht selbststoppend. Auch in diesem Falle würde das Programm mit seinem eigenen Programmtext als Eingabe zunächst das Programm "Stoptest" abarbeiten, wobei die boolesche Variable nun aber den Wert FALSE erhält. Das aber bedingt die Ausgabe "Das Programm ist nicht selbststoppend." und den anschließenden Stop des Programms, was wiederum im Widerspruch zur Annahme steht.
Das Programm "Seltsam trägt also völlig zu Recht seinen Namen, da es weder die Eigenschaft "selbststoppend" noch deren logische Negation besitzt. Da das aber nach den Gesetzen der Logik nicht möglich ist, kann ein solches Programm "Seltsam" nicht existieren. Dann kann es aber auch kein PASCAL-Programm "Stoptest" geben, da "Seltsam" aus dem Programm "Stoptest" konstruiert werden kann.
Damit ist dann vollständig bewiesen, daß es kein PASCAL-Programm gibt, mit dessen Hilfe man für jedes PASCAL-Programm entscheiden kann, ob es mit jeder beliebigen Eingabe nach endlich vielen Schritten stoppt oder nicht.

3. Formale Sprachen und abstrakte Automaten

Die Theorie der formalen Sprachen befaßt sich mit der Abstraktion von Alphabet, Wort, Grammatik und Sprache. Sie ist von Bedeutung für die Modellierung natürlicher Sprachen (automatische Spracherkennung) und für den Dialog Mensch - Maschine (Programmiersprachen, ihr "Verstehen" durch Automaten und automatische Übersetzung in Maschinensprachen mittels Compiler und Interpreter).
Betrachtet man natürliche Sprachen, so weiß jeder aus dem Sprachunterricht, daß jede Sprache eine Syntax, eine Semantik und eine Pragmatik hat.

Syntax	Menge der Vorschriften und Regeln, die den formal korrekten Aufbau von Wörtern und Sätzen angeben
Semantik	beschreibt die Bedeutung eines syntaktisch korrekten Satzes
Pragmatik	beschreibt die Wirkung, die ein Satz beim Nutzer auslöst, enthält folglich sehr stark

subjektive Aspekte individueller Nutzer

Eine erste Abstraktion besteht darin, daß bei formalen Sprachen nur noch die Syntax und die Semantik ausschlaggebend sind. Auch auf eine Unterscheidung von Wörtern und Sätzen, wie es bei natürlichen Sprachen üblich ist, wird verzichtet.

Definitionen

- Alphabet V $=_{def.}$ endliche, nichtleere Menge von unterscheidbaren Grundzeichen (oder Buchstaben), so daß jede Folge von Zeichen auf genau eine Art mittels Grundzeichen aus V darstellbar ist.
  V = { | , || } und V = { 1, 4, 41 } sind z. B. keine Alphabete
- w Wort über V $=_{def}$ w endliche Folge von Grundzeichen
- leeres Wort Λ $=_{def}$ enthält kein Grundzeichen
- $V^* =_{def}$ Menge aller Wörter über V
- Länge eines Wortes w über V: $|w| =_{def}$ Anzahl der Grundzeichen (mit Wiederholung) im Wort w (z. B. w = aaba $\Rightarrow$ |w| = 4)
- Sei $a \in V$: $a^k =_{def}$ aa...a (k-mal)
- Verkettung $w_1 \cdot w_2$ zweier Wörter $w_1, w_2 \in V^* =_{def}$ Wort w, das durch Hintereinanderschreiben von w_1 und w_2 entsteht
  Der Punkt zwischen w_1 und w_2 wird meist weggelassen. Ferner gilt:
  $w_1 \cdot w_2 \neq w_2 \cdot w_1$
  $w_1 \Lambda = \Lambda w_1$
  $|w_1 \cdot w_2| = |w_1| + |w_2|$

Definition: (formale Sprache)
Es sei V ein Alphabet. Eine formale Sprache (über V) ist dann eine beliebige Teilmenge von V*.

Präzisierungen sind möglich durch
a) Auflisten aller Elemente von L (bei unendlich vielen Elementen nicht machbar)
b) endliche Erzeugungsverfahren (Grammatiken: Angabe von Regeln, die alle Wörter von L erzeugen)
c) endliche Erkennungsverfahren (Automaten: Angabe eines Algorithmus, der entscheidet, ob $w \in L$)

Einige Autoren (z. B. Sander u.a.) sprechen nur dann von einer formalen Sprache L , wenn L durch ein endliches formales System vollständig beschreibbar ist.

3. 1. Grammatikmodelle

1956 entwickelte Noam Chomsky Grammatikmodelle als spezielle endliche Erzeugungsverfahren.

Definition: (Grammatik)

Eine Grammatik ist ein 4-Tupel $G = (V, \Sigma, R, S)$, das folgende Bedingungen erfüllt:

- V ist eine endliche Menge, die Menge der Variablen
- Σ ist eine endliche Menge, die Menge der Terminale ($\Sigma \cap V = \emptyset$)
  R ist eine endliche Menge von Ersetzungsregeln, formal eine endliche Teilmenge von $((V \cup \Sigma)^* \setminus \Sigma^*) \times (V \cup \Sigma)^*$
- $S \in V$ ist eine Startvariable

Seien $u, v \in (V \cup \Sigma)^*$. Wir definieren dann die Relation $u \Rightarrow_G v$ (in Worten: u geht unter G <u>in einem Schritt</u> in v über), falls u und v die Form haben

$$u = xyz$$
$$v = xy'z \qquad \text{mit } x, z \in (V \cup \Sigma)^*$$

und $(y, y') \in R$. Wenn klar ist, welche Grammatik G gemeint ist, dann schreiben wir einfach $u \Rightarrow v$ statt $u \Rightarrow_G v$.

Mit $\Rightarrow_G^*$ bezeichnen wir die reflexive und transitive Hülle von $\Rightarrow_G$, die wie folgt definiert ist:
$u \Rightarrow_G^* v$ (in Worten: u geht unter G in <u>endlich vielen Schritten</u> in v über), wenn $u = v$ oder Wörter $w_0, w_1, ..., w_n \in (V \cup \Sigma)^*$ existieren mit $w_0 = u$, $w_n = v$ und $w_i \Rightarrow_G w_{i+1}$ für $i = 0, 1, ..., n-1$.

Definition: (von einer Grammatik erzeugte Sprache)

Es sei $G = (V, \Sigma, R, S)$ eine beliebige Grammatik.
Die von G erzeugte Sprache ist die Menge $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$.

Wir nennen $S \Rightarrow_G^* w \in \Sigma^*$ auch eine Ableitung und $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow ... \Rightarrow w$ eine Ableitungsfolge für w. Anstelle von $(y, y') \in R$ schreiben wir auch $y \rightarrow y'$.

Beispiele:

1) $G = (\{S, T, F\}, \{+, -, *, /, a, (,)\}, R, S)$, wobei

$R = \{ S \rightarrow T,$
$S \rightarrow S+T,$
$S \rightarrow S-T,$
$S \rightarrow S/T$
$T \rightarrow F,$
$T \rightarrow T^* F,$
$F \rightarrow a,$
$F \rightarrow (S) \}$

Mit dieser Grammatik lassen sich alle korrekt geklammerten arithmetischen Ausdrücke darstellen. Es gilt z. B. a * a * (a - a) + a $\in$ L(G) bzw. a (a+a) $\notin$ L(G).

2) einfache deutsche Sätze

<Satz>	→	<Subjekt> <Prädikat> <Objekt>
<Subjekt>	→	<S_Artikel> <Substantiv>
<S_Artikel>	→	Die
<Artikel>	→	die
<Substantiv>	→	Katze
<Substantiv>	→	Maus
<Substantiv>	→	Ratte
<Prädikat>	→	jagt
<Prädikat>	→	frißt
<Objekt>	→	<Artikel> <Substantiv>

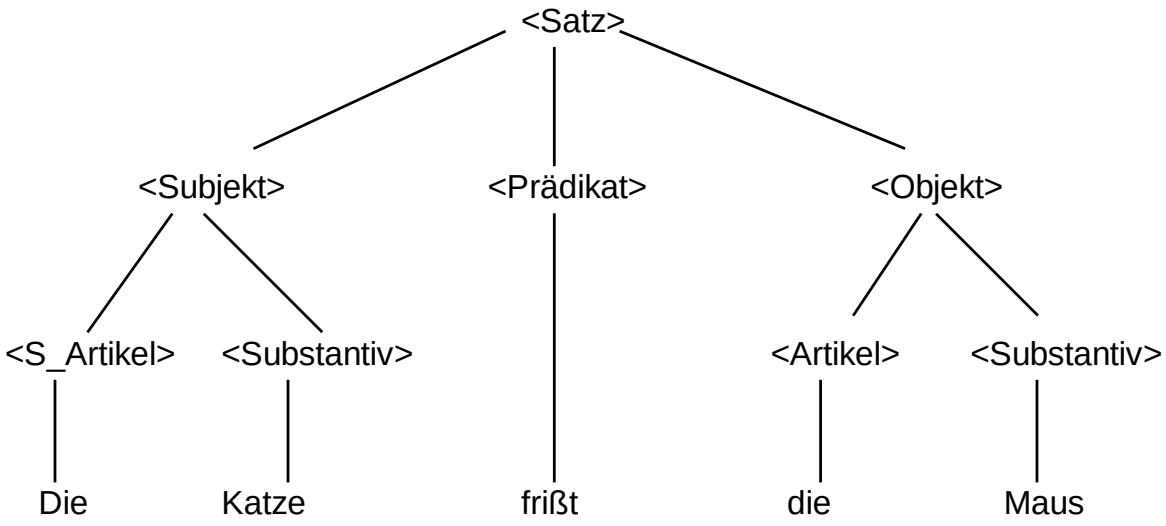
Hierbei sollen mit spitzen Klammern die Variablen deutlich gemacht werden, die Platzhalter für die entsprechendenTerminale sind.

Durch diese Grammatik können z. B. folgende Sätze abgeleitet werden:

Die Katze jagt die Maus
Die Maus frißt die Ratte
Die Katze frißt die Katze
u. ä.

Besonders anschaulich kann die Wirkungsweise einer Grammatik durch einen **Syntaxbaum** dargestellt werden.

Hierbei ist der Vaterknoten jeweils mit der linken Seite einer Regel beschriftet und seine Söhne sind die Objekte, die auf der rechten Seite der Regel stehen.



3) $L(G) = \{ a^n b^n c^n \mid n \in N \}$

Menge der Variablen: {S, B, C }

Menge der Terminale: {a, b, c }

Ersetzungsregeln:

S	→	aSBC
S	→	aBC
CB	→	BC
aB	→	ab
bB	→	bb
bC	→	bc
cC	→	cc

Es gilt dann zum Beispiel folgende Ableitungsfolge:

$S \Rightarrow aSBC \Rightarrow aaBCBC \Rightarrow aabCBC \Rightarrow aabBCC \Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow aabbcc$

Den Zusammenhang zu Programmiersprachen werden wir erst später herstellen. Ferner sei schon hier angemerkt, daß diese Sprache nicht durch eine kontextfreie Grammatik erzeugt werden kann.

Chomsky - Hierarchie

Es sei G = (V, Σ, R, S) eine beliebige Grammatik. Dann definieren wir

Definition (Chomsky-Grammatik vom Typ i)

Eine Grammatik G ist vom Chomsky - Typ

0 (**allgemein**) ⇔ für alle (u, v) ∈ R gilt: u ∈ (V∪Σ)* \ Σ* und v ∈ (V∪Σ)*.

1	(**kontextabhängig**),	⇔	für alle (u, v) ∈ R gilt: $u = u_1 A u_2$ und $v = u_1 w u_2$ mit $A \in V$, $u_1, u_2, w \in (V \cup \Sigma)^*$ und $w \neq \Lambda$.
2	(**kontextfrei**)	⇔	für alle (u, v) ∈ R gilt: $u \in V$, $v \in (V \cup \Sigma)^*$ und $v \neq \lambda$.
3	(**regulär**),	⇔	für alle (u, v) ∈ R gilt: $u \in V$ und $v \in \Sigma$ oder $v = wA$ mit $w \in \Sigma$ und $A \in V$.

Definition: (Typ - i - Sprache)
Eine Sprache L heißt Typ - i - Sprache, wenn eine Grammatik G vom Typ i existiert mit L(G) = L

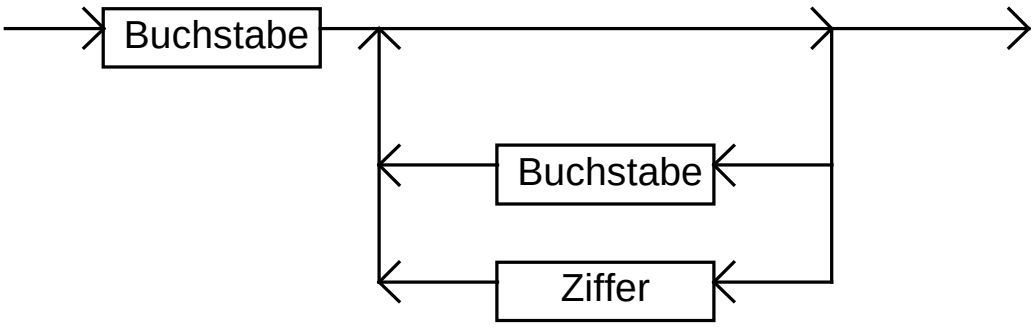
Anmerkungen:

- Da jede Typ - i - Grammatik ist erst recht eine Typ - (i-1) - Grammatik ist, ist auch jede Typ - i - Sprache eine Typ - (i-1) - Sprache.
- Für zahlreiche Anwendungen in der praktischen Informatik (Syntaxanalyse, Compilerbau) sind vor allem die Sprachen vom Typ 3 und 2 von Interesse. Deshalb wurden diese auch besonders intensiv untersucht. Nicht selten führen konkrete Fragesstellungen in der Praxis aber auch zu Typ 1 oder sogar zu Typ 0 Sprachen. Da diese algorithmisch schwieriger handhabbar sind, wird oft versucht so weit wie möglich doch mit kontextfreien Grammatiken zu arbeiten. Das wird zum Beispiel dadurch erreicht, daß die Fälle, die den Bereich des Kontextfreien zerstören, als Sonderfälle ausgegliedert werden.

Beispiele:

1) Erzeugung zulässiger Bezeichner in PASCAL

In der Vorlesung Algorithmen und Programmierung I haben Sie gelernt, daß Bezeichner in der Programmiersprache PASCAL Zeichenketten sind, die nur aus Buchstaben und Ziffern bestehen und mit einem Buchstaben beginnen. Die korrekte Bildung von Bezeichnern läßt sich mit sogenannten **Syntaxdiagrammen** wie folgt beschreiben:



Um die Notation der vielen Regeln zu verkürzen, ist die sogenannte BACKUS-NAUR-Form zur Darstellung einer Grammatik vorteilhaft, die speziell für kontextfreie Grammatiken entwickelt wurde. Dabei wird anstelle von mehreren Regeln, die alle dieselbe linke Seite haben, nur eine einzige Regel unter Verwendung eines senkrechten Striches geschrieben, der die alternativen

rechten Seiten trennt.
Mit spitzen Klammern sollen wiederum die Variablen deutlich gemacht werden, die Platzhalter für die entsprechendenTerminale sind.
Die Regeln einer alle zulässigen Bezeichner erzeugenden (kontextfreien) Grammatik lauten dann:

<Bezeichner> → <Buchstabe> | <Bezeichner> <Buchstabe> | <Bezeichner> <Ziffer>
<Buchstabe> → a | b | ... | z | A | B | ... | Z |
<Ziffer> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

2) G = (V, Σ , R, S) mit

V = {S}
Σ = {a, b}
und R die Menge folgender Regeln:

(1) S → ab
(2) S → aSb

Offensichtlich ist die Grammatik G kontextfrei. Sie erzeugt die Sprache $L = \{a^n b^n \mid n \in N\}$.
Diese Sprache hat ebenfalls in PASCAL eine große Bedeutung, denn sie stellt die Klammerstruktur in einfachster Form dar. Denken Sie an arithmetische Ausdrücke mit den Klammern "(" und ")" , an Verbundanweisungen mit den Klammern "BEGIN" und "END" oder an Wiederholungen mit Endab-frage mit den Klammern "REPEAT" und "UNTIL".

3) Die Grammatik im Beispiel 2 auf Seite 21 ist kontextabhängig (Typ 1).

4) G = (V, Σ , R, S) mit

V = {S, A, B}
Σ= {0, 1}
und R die Menge folgender Regeln

(1) S → 0A
(2) S → 1B
(3) A → 0A
(4) A → 0S
(5) A → 1B
(6) B → 1B
(7) B → 1
(8) B → 0
(9) S → 0

Diese Grammatik ist regulär (Typ 3). Eine mögliche Ableitungsfolge ist z. B.
S ⇒ 0A ⇒ 00A ⇒ 000A ⇒ 0001B ⇒ 00010
Welche Sprache wird durch diese Grammatik erzeugt?

Definition
Zwei Grammatiken G_1 und G_2 heißen äquivalent, wenn $L(G_1) = L(G_2)$.

L_i bezeichne die Menge aller Sprachen vom Typ i. Dann gilt:

Satz:
$L_3 \subseteq L_2 \subseteq L_1 \subseteq L_0$

Teile dieser Aussage werden wir im Zusammenhang mit der Leistungsfähigkeit abstrakter Automaten beweisen.

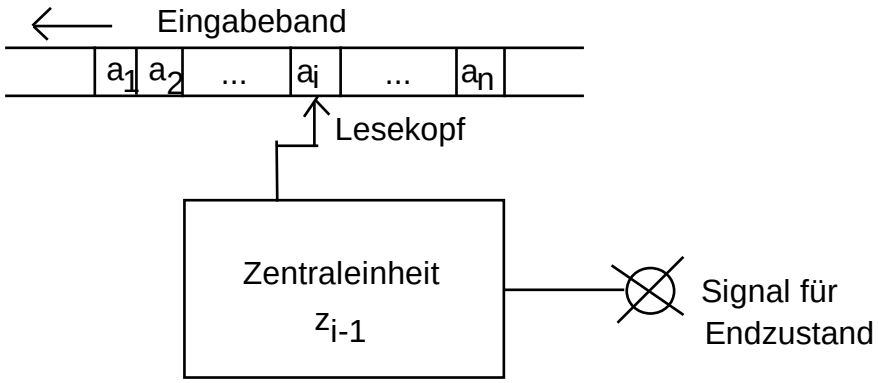
Satz:
Wenn $L \in L_1$, dann ist L entscheidbar.

3. 2. Abstrakte Automaten

Abstrakte Automaten gibt es mit oder ohne Ausgabe. Ohne Ausgabe werden sie auf Eingabewörter angesetzt, um zu "erkennen" (oder "zu akzeptieren"), ob ein Wort $w \in L$ ist. Die Menge der akzeptierten Wörter bildet dann die durch den Automaten definierte Sprache. Der Mechanismus derartiger endlicher Erkennungsverfahren ist in gewisser Weise das Gegenstück zu den endlichen Erzeugungsverfahren in Form von Grammatiken.

3. 2. 1. Endliche Automaten

Endliche Automaten begegnen uns sehr zahlreich in der Praxis. Sie bestehen aus einer Zentraleinheit, die endlich viele verschiedene innere Zustände annehmen kann, einem Eingabeband, das schrittweise um eine Position weiter nach links rücken kann und einem Lesekopf, der die auf dem Eingabeband stehenden Zeichen (Buchstaben) lesen kann.



Formal können sie wie folgt definiert werden:

Definition (endlicher Automat)

Ein endlicher Automat EA ist ein 5-Tupel

$EA = (X, Z, \delta, z_0, E)$, wobei

X	ein endliches Eingabealphabet
Z	eine endliche Menge von Zuständen, wobei $X \cap Z = \emptyset$.
$z_0 \in Z$	ein sogenannter Startzustand
$E \subseteq Z$	eine nichtleere Menge von Endzustände
$\delta: X \times Z \rightarrow Z$	die Überführungsfunktion, die jedem Paar (a, z) mit $a \in X$ und $z \in Z$ einen Zustand $z' \in Z$ zuordnet.

Ein endlicher Automat EA erkennt (akzeptiert) ein Wort $w = a_1 a_2 ... a_n \in X^*$ dadurch, daß der Automat das Eingabewort von links nach rechts buchstabenweise liest und dabei eine Folge von Zuständen $z_0, z_1, ..., z_n$ durchläuft. Hierbei ist z_0 der Startzustand, $z_i = \delta(a_i, z_{i-1})$, und z_n muß in E liegen, also ein Endzustand sein.

Die Funktion δ, die oben bisher nur für das Lesen von Zeichen erklärt ist, kann wie folgt rekursiv zu einer Wortfunktion $\delta^*: X^* \times Z \rightarrow Z$ fortgesetzt werden:

$\delta^*(\lambda, z) = z$

$\delta^*(wx, z) = \delta(x, \delta^*(w,z))$ für $x \in X$, $w \in X^*$ und $z \in Z$.

Um den neuen Zustand $\delta^*(wx, z)$ zu ermitteln, in den der Automat bei Eingabe von wx im Zustand z gelangt, bestimmen wir den Zustand $\delta^*(w, z)$, in den der Automat durch Abarbeiten des Teilwortes w im Zustand z gelangt, und bearbeiten in diesem neuen Zustand das Zeichen x. Wir gelangen folglich in den Zustand $\delta(x, \delta^*(w,z))$.

Definition

Es sei $EA = (X, Z, \delta, z_0, E)$ ein endlicher Automat. Dann ist $L(EA) = \{w \in X^* \mid \delta^*(w, z_0) \in E\}$ die Menge aller von EA akzeptierten Wörter.

Diese endlichen Automaten ohne Ausgabe können lediglich ein Wort akzeptieren, aber keine Ausgabe produzieren. In der Fachliteratur werden sie deshalb häufig auch **Akzeptoren** genannt.

Beispiele:

1) Es sei $EA = (X, Z, \delta, z_0, E)$ mit

X = Menge aller druckbaren ASCII-Zeichen (Buchstaben, Ziffern, Sonderzeichen)
$Z = \{z_0, z_k, z_f\}$
$E = \{z_k\}$

und δ vereinfacht durch folgende Tabelle beschrieben:

	B	Z	S
z_0	z_k	z_f	z_f
z_k	z_k	z_k	z_f
z_f	z_f	z_f	z_f

Dabei stehe B für alle Buchstaben, Z für alle Ziffern und S für alle Sonderzeichen.
Der Automat akzeptiert z. B. das Wort BAUM, aber nicht das Wort 7Z.

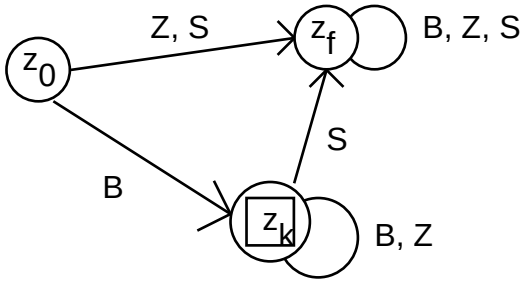
Zeitpunkt	**Zustand**	**Eingabezeichen**
0	z_0	B
1	z_k	A
2	z_k	U
3	z_k	M
4	z_k	Λ **Stop!!**

Aber:

Zeitpunkt	**Zustand**	**Eingabezeichen**
0	z_0	7
1	z_f	Z
2	z_f	Λ **Stop!!**

Offensichtlich ist die von M akzeptierte Menge wiederum die Menge aller korrekt gebildeten Bezeichner in PASCAL.

Für die Darstellung eines endlichen Automaten bieten sich ferner sogenannte Zustandsgraphen an. Jedem Zustand entspricht dabei ein Knoten, und dem Übergang von einem Zustand z_i in einen Zustand z_j entspricht eine gerichtete Kante (z_i, z_j). Wenn $z_j = \delta(a, z_i)$, dann erhält die Kante (z_i, z_j) die Bezeichnung a.
Für den obigen Automaten sieht der Zustandsgraph wie folgt aus:



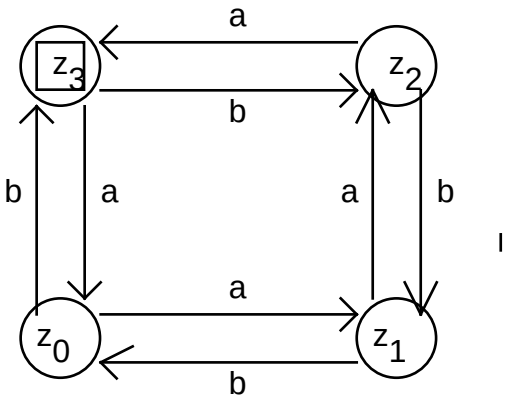
2) Sei EA = (X, Z, δ, z_0, E) mit

X = {a, b}
Z = {z_0, z_1, z_2, z_3}
E = {z_3}

und δ wie folgt definiert:

δ (a, z_0) = z_1,
δ (b, z_0) = z_3,
δ (a, z_1) = z_2,
δ (b, z_0) = z_0,
δ (a, z_2) = z_3,
δ (b, z_2) = z_1,
δ (a, z_3) = z_0,
δ (b, z_3) = z_2

Der zugehörige Zustandsgraph sieht dann wie folgt aus



Offensichtlich ist $a^9b^2 \in L$, denn der Automat arbeitet dieses Wort wie folgt ab:

Zeitpunkt	**Zustand**	**Eingabezeichen**
0	z_0	a
1	z_1	a
2	z_2	a

3	z_3	a
4	z_0	a
5	z_1	a
6	z_2	a
7	z_3	a
8	z_0	a
9	z_1	b
10	z_0	b
11	z_3	Λ **Stop!!**

Es leuchtet sofort ein, daß dieser Automat z. B. auch das Wort a^5b^2 akzeptiert., dann nach fünf a's befindet sich der Automat auch im Zustand z_0. Insgesamt kann gezeigt werden, daß dieser Automat die Sprache L = {w ∈ X* | ((Anzahl a's in w) - (Anzahl b's in w)) MOD 4 = 3}erkennt .

Für viele Anwendungen ist es günstig, neben den deterministischen endlichen Automaten auch noch nichtdeterministische endliche Automaten zu betrachten. Wir werden später sehen, daß diese sich in der Leistungsfähigkeit nicht unterscheiden, aber viele Überlegungen können dadurch vereinfacht werden.

Definition (nichtdeterministische endliche Automaten)
Ein nichtdeterministischer endlicher Automat ist ein 5-Tupel (X, Z, δ, z_0, E) mit X, Z, z_0 und E wie oben bei deterministischen endlichen Automaten definiert und δ eine Abbildung von X x Z in die Potenzmenge von Z.
Ein Wort w = wird von einem nichtdeterministischen endlichen Automaten akzeptiert, wenn es eine
endliche Folge von Zuständen z_0, z_1, ..., z_n gibt, so daß z_0 Startzustand, $z_i \in E \cap \delta(x_i, z_{i-1})$ und $z_n \in E \cap \delta(w, z_0)$ ist.

Der entscheidende Unterschied zwischen deterministischen und nichtdeterministischen endlichen Automaten besteht also darin, daß δ (x, z) anstelle eines einzigen Folgezustandes eine (möglicherweise auch leere) Menge von Folgezuständen liefert.
Da deterministische endliche Automaten Spezialfälle von nichtdeterministischen endlichen Automaten sind, gibt es zu jeder von einem deterministischen Automaten akzeptierten Sprache L trivialerweise einen nichtdeterministischen Automaten, der die Sprache L akzeptiert. Es gilt aber auch der folgende Satz

Satz: (ohne Beweis)
Sei L eine von einem nichtdeterministischen endlichen Automaten akzeptierte Sprache. Dann gibt es stets auch einen deterministischen endlichen Automaten, der L akzeptiert.

Wir werden beim Beweis des folgenden Satzes davon Gebrauch machen:

Satz:
Zu jeder regulären Grammatik G gibt es einen endlichen Automaten M, der die von G erzeugte Sprache akzeptiert, d.h. L(G) = L(M)

Beweis:

Es sei G = (V, Σ , R, S) eine reguläre Grammatik mit Regeln der Form

$A \rightarrow aB$ oder

$A \rightarrow a$, wobei $A, B \in V$ und $a \in \Sigma$

Dazu konstruieren wir folgenden nichtdeterministischen endlichen Automaten EA = (X, Z, δ, z_0, E):

$X = 3$

$Z = V \ \{z_e\}$

$E = \{z_e\}$

$z_0 = S$ und

$\delta(a, A) = B$, falls $A \rightarrow aB$ in R

$\delta(a, A) = z_e$, falls $A \rightarrow a$ in R.

Dann gilt L(EA) = L(G). Statt zu untersuchen, ob $w \in L(G)$ ist, wird geprüft, ob der zu G gehörige Automat EA das Wort w akzeptiert, d. h. es wird geprüft, ob $\delta(w, z_0) \in E$ ist.

Beispiel:

Betrachten wir noch einmal die reguläre Grammatik im Beispiel 4 von Kapitel 3. 1.

G = (V, Σ, R, S) mit

V = {S, A, B},

Σ = {0, 1}

und R die Menge folgender Regeln

(1)	$S \rightarrow 0A$	(6)	$B \rightarrow 1B$
(2)	$S \rightarrow 1B$	(7)	$B \rightarrow 1$
(3)	$A \rightarrow 0A$	(8)	$B \rightarrow 0$
(4)	$A \rightarrow 0S$	(9)	$S \rightarrow 0$
(5)	$A \rightarrow 1B$		

Der zugehörige (nichtdeterministische) endliche Automat EA = (X, Z, δ, S, E) sieht wie folgt aus:

X = {0, 1},

$Z = \{S, A, B, z_e\}$

$E = \{z_e\}$ und für δ gilt:

$\delta(0, S) = \{A, z_e\}$, denn

$S \rightarrow 0A$	entspricht	$\delta(0, S) = A$
$S \rightarrow 0$	entspricht	$\delta(0, S) = z_e$

$\delta(0, A) = \{A, S\}$, denn

$A \rightarrow 0A$	entspricht	$\delta(0, A) = A$
$A \rightarrow 0S$	entspricht	$\delta(0, A) = S$

$\delta(1, B) = \{B, z_e\}$, denn

$B \rightarrow 1B$	entspricht	$\delta(1, B) = B$
$B \rightarrow 1$	entspricht	$\delta(1, B) = z_e$

$\delta(1, S) = \{B\}$, denn

$S \rightarrow 1B$	entspricht	$\delta(1, S) = B$

$\delta(1, A) = \{B\}$, denn

$A \rightarrow 1B$	entspricht	$\delta(1, A) = B$

$\delta(0, B) = \{z_e\}$, denn

$B \rightarrow 0$	entspricht	$\delta(0, B) = z_e$

und $*(x, z_e) = \dot{\imath}$ für $x \in \{0, 1\}$

Die Grammatik ist offensichtlich regulär, und eine mögliche Ableitungsfolge ist z. B.

$S \Rightarrow 0A \Rightarrow 00A \Rightarrow 000A \Rightarrow 0001B \Rightarrow 00010$.

Dieser Ableitungsfolge entspricht folgender Akzeptierungsprozeß des endlichen Automaten für das Wort w = 00010. Zum besseren Verständnis werde ich den Zustand der aktuellen Zustandsmenge, mit dem ich im nächsten Schritt weiterarbeite, unterstreichen

Zeitpunkt	**Zustandsmenge**	**Eingabezeichen**
0	$\{\underline{S}\}$	0
1	$\{\underline{A}, z_e\}$	0
2	$\{\underline{A}, S\}$	0
3	$\{\underline{A}, S\}$	1
4	$\{\underline{B}\}$	0
5	$\{z_e\}$	Λ **Stop!!**

Aber auch die Umkehrung dieses Satzes gilt:

Satz:
Zu jedem endlichen Automaten $EA = (X, Z, \delta, z_0, E)$ gibt es eine reguläre Grammatik G, die die von EA erkannte Sprache erzeugt, d.h. $L(G) = L(EA)$

Beweis:

Es sei $G = (V, \Sigma, R, S)$ mit

$V = Z$ und

$S = z_0$

Wenn $\lambda \in L(EA)$ (d. h. falls $z_0 \in E$), dann enthält R die Regel

$z_0 \rightarrow \lambda$

Ferner besteht R aus folgenden Regeln:

$z_1 \rightarrow xz_2$, falls $\delta(x, z_1) = z_2$ und $z_2 \neq E$

$z_1 \rightarrow x$, falls $\delta(x, z_1) = z_2$ und $z_2 \in E$

Nun gilt für alle $w = x_1x_2 \ldots x_n \in X^*$:

$w \in L(EA)$ genau dann, wenn es eine Folge von Zuständen $z_0, z_1, \ldots, z_n$ gibt mit:
z_0 ist Startzustand, $z_n \in E$ und für i = 1, .., n gilt: $\delta(x_i, z_{i-1}) = z_i$

genau dann, wenn es eine Folge von Variablen $z_0, z_1, \ldots, z_n$ gibt mit:
z_0 ist Startvariable und es gilt : $z_0 \Rightarrow x_1z_1 \Rightarrow x_1x_2z_2 \Rightarrow x_1x_2\ldots x_{n-1}z_{n-1} \Rightarrow x_1x_2\ldots x_{n-1}x_n$

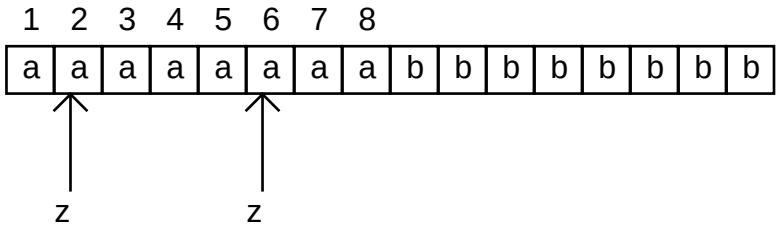
Satz:
Es gibt keinen endlichen Automaten, der genau die Sprache $\{a^n b^n \mid n \in N\}$ erkennt.

Beweis:

Angenommen, es existiert ein endlicher Automat mit dem Eingabealphabet {a, b} und m Zuständen, der genau die Wörter der Sprache $L = \{a^n b^n \mid n \in N\}$ erkennt. Wir zeigen dann, daß im Wider-spruch zu dieser Annahme der Automat zusätzlich Wörter erkennt, die nicht zu L gehören. Der Automat hat endlich viele Zustände. Wir wählen nun ein Wort $a^k b^k$ der Sprache mit $k > m$ aus. Beim Abarbeiten des Teilwortes a^k kann der Automat nicht k verschiedene Zustände durchlaufen, da es ja nur m verschiedene Zustände gibt und $m < k$ ist. Es wird also mindestens ein Zustand z zweimal durchlaufen. Beim Abarbeiten des Teilwortes a^k durchläuft der Automat also wenigstens einmal einen Zyklus vom Zustand z wieder in den Zustand z. Dieser Zyklus habe o.B.d.A. die Länge l. Das bei diesem Zyklus durchlaufende Teilwort a^l von a^k kann aber dann herausgeschnitten bzw. beliebig oft in das Wort a^k hineinkopiert werden, so daß der Automat auch beim Abarbeiten des Wortes
$a^{k-l}b^k$, $a^{k+l}b^k$ usw. in denselben Endzustand gelangt.

Widerspruch!!

Veranschaulichen wir uns die Beweisidee an einem Beispiel mit $k = 8$ und $l = 4$:
Ohne Beschränkung der Allgemeinheit nehmen wir an, daß der Automat beim zweiten und beim sechsten 'a' im Zustand z ist.



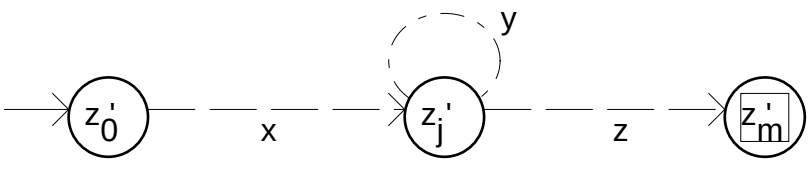
Schneiden wir das Teilwort aaaa von Position 2 bis Position 5 heraus oder kopieren wir es nochmals an Position 2 oder 6 in das Wort hinein, so arbeitet der Automat wie vorher weiter. Der Automat gelangt daher auch nach Abarbeitung von a^4b^8 und $a^{12}b^8$ in denselben Endzustand wie bei a^8b^8. Entsprechend akzeptiert er auch $a^{16}b^8$, $a^{20}b^8$, ...

Dieser Beweisgedanke läßt sich verallgemeinern und führt zu dem sogenannten Pumping-Lemma für endliche Automaten.

Satz (Pumping-Lemma für Sprachen endlicher Automaten):
Es sei EA = (X, Z, δ, z_0, E) ein endlicher Automat mit n Zuständen (|Z| = n). Dann ist für jedes Wort $w \in L(EA)$ mit einer Länge $|w| \geq n$ die folgende Forderung erfüllt:
w läßt sich schreiben als w = xyz (x, y, z $\in$ X*) mit den folgenden Eigenschaften:

- $|xy| \leq n$,
- $|y| \geq 1$ und
- $\forall i \in N$ gilt: $xy^iz \in L(EA)$.

Beweis:
Wir betrachten ein beliebiges Wort $w = x_1'x_2'...x_m'$ der Sprache L(EA), bestehend aus m Zeichen des Eingabealphabets X, wobei $m \geq n$ sei. Wir nehmen ferner an, daß der Automat beim Akzeptieren dieses Wortes die Zustandsfolge z_0', z_1', ..., z_m' mit $z_0' = z_0$, $z_i' \in Z$ und $z_m' \in E$ durchläuft. Dies sind insgesamt m+1 Zustände. Da $m \geq n$ ist und nach Voraussetzung nur n verschiedene Zustände existieren, muß mindestens einer dieser Zustände in der Folge der ersten n+1 Zustände doppelt auftreten. Wir nehmen also an, daß $z_j' = z_k'$ gilt mit $0 \leq j < k \leq n$. Setzen wir nun $x = x_1'...x_j'$,
$y = x_{j+1}'...x_k'$ und $z = x_{k+1}'...x_m'$, dann ist offensichtlich $w = xyz$ mit $|xy| = k \leq n$ und $|y| = k - j \geq 1$. Außerdem gilt, wie in der folgenden Abbildung verdeutlicht $\delta(z_0, xy) = \delta(z_0, xyz) = \delta(z_0, xy^iz) = z_m' \in E$, d. h. es gilt: $\forall i \in N : xy^iz \in L(EA)$. qed



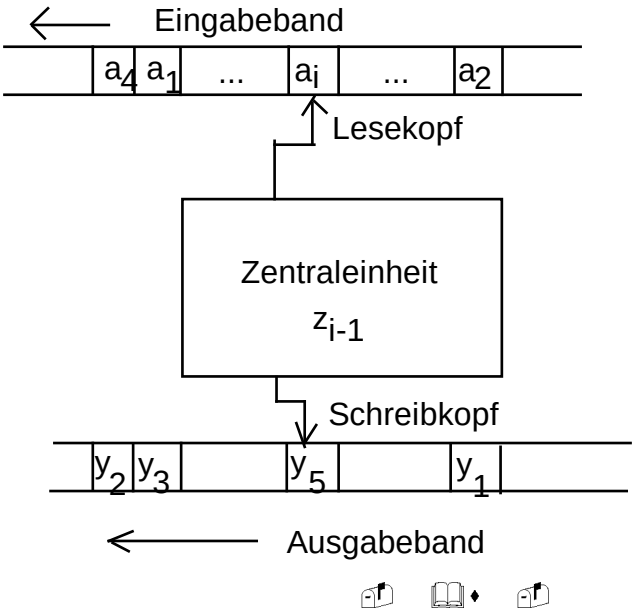
Der obige Satz heißt Pumping-Lemma, weil jedes Wort der Sprache, das aus mindestens n Zeichen besteht, sich zu längeren Wörtern der Sprache "aufpumpen" läßt. Bei vielen Beweisen zur Nichtakzeptierung gewisser Sprachen wird dieses Lemma benutzt.

Beispiel:
Sei $L = \{a^k \mid k = i^2 \text{ mit } i \in N\}$ die Menge aller Wörter über dem Alphabet {a}, deren Länge eine Quadratzahl ist. Wir zeigen, daß L nicht die Sprache eines endlichen Automaten sein kann.

Beweis:
Nehmen wir dazu an, es gäbe einen endlichen Automaten $EA = (X, Z, \delta, z_0, E)$ mit n Zuständen, der L akzeptiert. Dann müßte sich jedes Wort w 0 L mit einer Wortlänge größer oder gleich n gemäß dem Pumping-Lemma zerlegen lassen. Betrachten wir speziell das Wort w der Länge n^2 und eine Zerlegung $w = xyz$ mit $1 \leq |y| \leq n$ und $xy^iz \in L$ für alle i.
Die letzte Beziehung gilt für alle i, also auch für i = 2, d. h. $xyyz \in L$. Dann gilt aber:
(1) xyyz hat mindestens die Länge n^2+1, da $|xyyz| = |xyz| + |y| \geq n^2+1$
(2) xyyz hat höchstens die Länge n^2+n, wobei $n^2+n < (n+1)^2$
Somit folgt $n^2 < |xyyz| < (n+1)^2$. Das bedeutet aber, daß die Länge von xyyz keine Quadratzahl ist, und folglich gehört xyyz nicht zu L. Das aber ist ein Widerspruch zur Annahme. qed

Endliche Automaten mit Ausgabe

Im Gegensatz zu oben definierten endlichen Automaten besitzen endliche Automaten mit Ausgabe zusätzlich ein Ausgabeband, das schrittweise um eine Position weiter nach links rücken kann und einen Schreibkopf, der Zeichen des Ausgabealphabets auf das Ausgabeband schreiben kann. Auf eine Menge von Endzuständen wird dafür verzichtet.



In der Fachliteratur werden diese Automaten **endliche Mealy-Automaten** genannt. Formal können sie wie folgt definiert werden:

Definition (endlicher Mealy-Automat)

Ein endlicher Mealy-Automat EMA ist ein 6-Tupel

$$EMA = (X, Z, Y, \delta, \lambda, z_0), \text{ wobei}$$

X	ein endliches Eingabealphabet
Z	eine endliche Menge von Zuständen, wobei $X \cap Z = \varnothing$.
Y	ein endliches Ausgabealphabet
$\delta: X \times Z \rightarrow Z$	die Überführungsfunktion, die jedem Paar (a, z) mit $a \in X$ und $z \in Z$ einen Zustand $z' \in Z$ zuordnet
$\lambda: X \times Z \rightarrow Y$	die Ausgabefunktion, die jedem Paar (a, z) mit $a \in X$ und $z \in Z$ ein Zeichen $y \in Y$ zuordnet
$z_0 \in Z$	ein sogenannter Startzustand

Ein endlicher Mealy-Automat EMA ist kein Akzeptor, bei dem die Klassifizierung der Eingabewörter im Vordergrund steht, sondern es geht darum, welche Ausgabewörter aus Y* bei Verarbeitung von Eingabewörtern aus X* erzeugt werden. Er stellt also Funktionen dar, die Wörtern aus X* in eindeutiger Weise Wörter aus Y* zuordnen.
Die Funktionen δ und λ, die oben bisher nur für Zeichen erklärt ist, können wie folgt rekursiv zu Wortfunktionen $\delta^*: X^* \times Z \rightarrow Z$ bzw. $\lambda^*: X^* \times Z \rightarrow Y^*$ fortgesetzt werden:

(1) $\delta^*(\Lambda,z) = z$

$\delta^*(wx, z) = \delta(x, \delta^*(w,z))$ für $x \in X$, $w \in X^*$ und $z \in Z$.

(2) $\lambda^*(\Lambda,z) = \Lambda$

$\lambda^*(wx,z) = \lambda^*(w,z)\, \lambda(x, \delta^*(w,z))$ für $x \in X$, $w \in X^*$ und $z \in Z$.

Bei Eingabe des leeren Wortes wird der Zustand z beibehalten , und es erfolgt keine Ausgabe.
Um den neuen Zustand $\delta^*(wx, z)$ zu ermitteln, in den der Automat bei Eingabe von wx im Zustand z gelangt, bestimmen wir den Zustand $\delta^*(w, z)$, in den der Automat durch Abarbeiten des Teilwortes w im Zustand z gelangt, und bearbeiten in diesem neuen Zustand das Zeichen x. Wir gelangen folglich in den Zustand $\delta(x, \delta^*(w,z))$.
Um das Ausgabewort zu ermitteln, das der Automat bei Eingabe von wx im Zustand z erzeugt, bestimmen wir zunächst das Ausgabewort $\lambda^*(w,z)$, das der Automat durch Abarbeiten des Teilwortes w im Zustand z erzeugt, und hängen daran das Zeichen an, das beim Lesen der Eingabe

x entsteht. Der aktuelle Zustand ist zu diesem Zeitpunkt $\delta^*(w,z)$. Das auszugebende Zeichen ist folglich $\lambda(x, \delta^*(w,z))$.

Der endlicher Mealy-Automat EMA = $(X, Z, Y, \delta, \lambda, z_0)$ berechnet offensichtlich die Funktion

$$f: X^* \rightarrow Y^* \text{ mit } f(w) = \lambda^*(w,z_0) \text{ für alle } w \in D(f).$$

Beispiele:

1) Fahrkartenautomat

Wir nehmen zur Vereinfachung an, daß es sich um einen Automaten für genau eine Sorte von Fahrkarten zum Preis von 2, 50 DM handelt. Der Automat nimmt Münzen im Wert von 0,50 DM und 1,00 DM an. Er besitzt außerdem eine Taste für Geldrückgabe und für die Fahrkartenausgabe. Zuviel eingeworfenes Geld wird nicht zurückgegeben.
Das Verhalten des Automaten können wir wie folgt beschreiben:

- Es sind vier verschiedene Eingaben möglich
  0, 50 DM-Münze einwerfen (x_{50})
  1, 00 DM-Münze einwerfen(x_{100})
  Taste für Geldrückgabe drücken (x_G)
  Taste für Fahrkartenausgabe drücken (x_F)
- Es sind folgende sechs interne Zustände möglich
  kein Geld eingeworfen (z_0)
  0, 50 DM eingeworfen (z_{50})
  1, 00 DM eingeworfen (z_{100})
  1, 50 DM eingeworfen (z_{150})
  2, 00 DM eingeworfen (z_{200})
  2, 50 DM oder mehr eingeworfen, d. h. Fahrkartenkauf möglich (z_F)
- Es sind drei verschiedene Ausgaben möglich
  nichts (y_n)
  Fahrkarte für 2, 50 DM (y_F)
  Rückgabe des gesamten eingeworfenen Geldes (y_R)

Seine Arbeitsweise kann wie folgt in einer Tabelle beschrieben werden, die sowohl die Überführungsfunktion als auch die Ausgabefunktion vollständig darstellt:

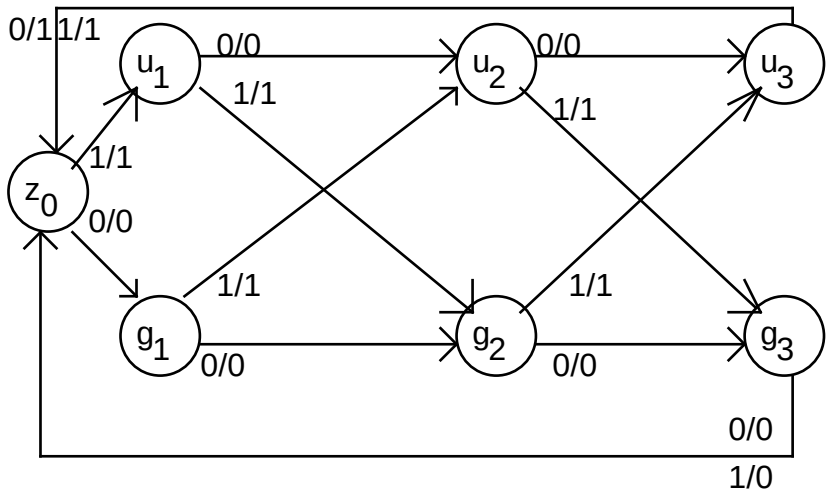
	z_0	z_{50}	z_{100}	z_{150}	z_{200}	z_F
x_{50}	z_{50}, y_n	z_{100}, y_n	z_{150}, y_n	z_{200}, y_n	z_F, y_n	z_F, y_n
x_{100}	z_{150}, y_n	z_{150}, y_n	z_{200}, y_n	z_F, y_n	z_F, y_n	z_F, y_n
x_G	z_0, y_R	z_0, y_R	z_0, y_R	z_0, y_R	z_0, y_R	z_0, y_R
x_F	z_0, y_n	z_{50}, y_n	z_{100}, y_n	z_{150}, y_n	z_{200}, y_n	z_0, y_F

2) Paritätsprüfmaschine

Bei einer seriellen Datenübertragung kommen Fehler vor, indem statt des Bits 0 das Bit 1 gesendet wird oder umgekehrt. Um festzustellen, ob die Übermittlung fehlerfrei erfolgt, sendet man nach jeweils einer festen Anzahl von Bits, dem eigentlichen Datenwort, ein **Paritätsbit** als Prüfbit. Es wird z. B. 0 gesetzt, falls die Anzahl der Einsen im Datenwort gerade ist, und 1, falls sie ungerade ist. Dies nennt man **gerade Parität**.

Nehmen wir an, daß jedes Datenwort aus drei Bits besteht und das folgende Bit immer als Trennzeichen zwischen Datenworten dient. Dieses Trennzeichen soll durch das Paritätsbit ersetzt werden. Wir wollen ferner festlegen, daß die Übertragung immer mit einem Datenwort beginnt.

Es sei $X = \{0, 1\}$, $Y = \{0, 1\}$, $Z = \{z_0, g_1, g_2, g_3, u_1, u_2, u_3\}$ und die Überführungsfunktion δ bzw. die Ausgabefunktion λ durch folgenden Graphen definiert:



In der Beschriftung der Kanten ist die erste Komponente das gelesene Zeichen und die zweite Komponente das auszugebende Zeichen. Die Beschriftung 0/1 an der Kante von u_3 nach z_0 bedeutet also, daß $\delta(0, u_3) = z_0$ und $\lambda(0, u_3) = 1$ ist. Da die Kante außerdem die Beschriftung 1/1 hat, gilt ferner $\delta(1, u_3) = z_0$ und $\lambda(0, u_3) = 1$.

Im folgenden wenden wir uns wieder Automatenmodellen zu, die als Akzeptoren wirken.

Daß ein endlicher Automat nicht die Sprache $\{a^n b^n \mid n \in N\}$ akzeptiert, bedeutet aus praktischer Sicht, daß kein endlicher Automat beliebig tief geschachtelte Klammerstrukturen in algebraischen Ausdrücken oder beliebig tief geschachtelte BEGIN-END-Strukturen in PASCAL erkennen kann. Für die Syntaxanalyse von PASCAL-Programmen ist dieses Automatenmodell daher nicht ausreichend.

3. 2. 2. Kellerautomat

Intuitiv kann ein endlicher Automat eine Sprache wie

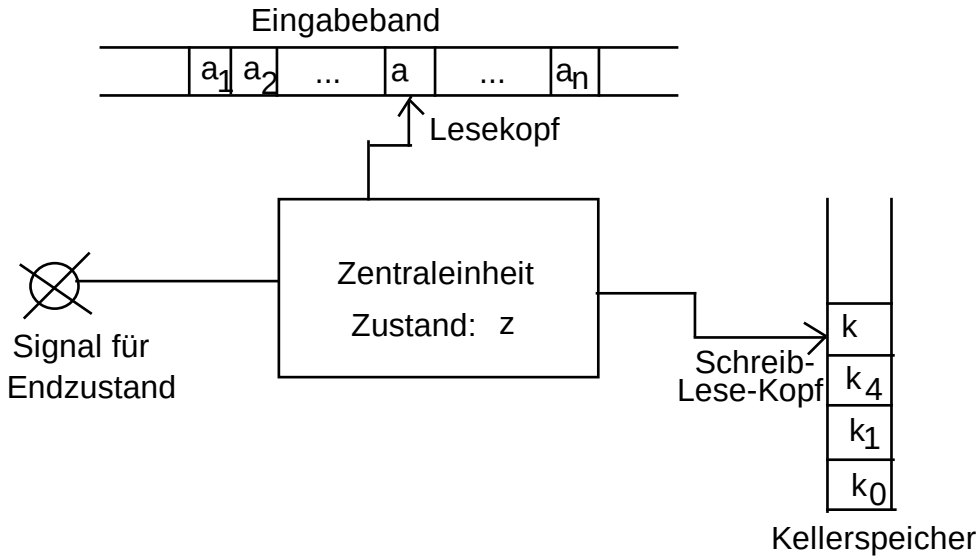
$$L = \{a^n b^n \mid n \in N\}$$

deshalb nicht erkennen, weil er zum Zeitpunkt, wenn er das Eingabezeichen 'b' erreicht, nicht mehr wissen kann, wie oft das Zeichen 'a' vorkam. Die einzige gespeicherte Information, die ihm zur Verfügung steht, ist der Zustand, in dem er sich befindet.

Eine naheliegende Erweiterung besteht deshalb darin, einen endlichen Automaten mit einem Speicher zu versehen, der es erlaubt, die Vergangenheit der Verarbeitung eines Wortes in gewissem Umfange festzuhalten.

Wir betrachten dazu zunächst einen sogenannten **Kellerspeicher** oder auch **Keller**. Ein Keller arbeitet nach dem **LIFO-Prinzip** (LIFO = last in - first out), d. h. stellt man sich den Kellerinhalt als senkrecht notierte endliche Folge von Zeichen vor, so kann nur auf das oberste Element des Keller-inhaltes zugegriffen werden (Zeichen anfügen, löschen oder lesen).

Mögliche Aktionen hängen jetzt nicht nur vom Zustand und vom gelesenen Eingabezeichen ab, sondern auch vom obersten Kellerzeichen. Bildlich kann man sich einen Kellerautomaten wie folgt darstellen:



Ein Verarbeitungsschritt besteht darin, daß der Automat das Zeichen unter dem Lesekopf des Eingabebandes und unter dem Schreib-Lese-Kopf des Kellers liest und - in Abhängigkeit vom aktuellen Zustand - seinen Zustand ändert und das oberste Kellerzeichen durch eine Folge von Zeichen ersetzt. Anschließend wird der Lesekopf um eine Position nach rechts und der Schreib-Lese-Kopf auf das oberste Kellerzeichen positioniert.

Definition (deterministischer Kellerautomat)

Ein (deterministischer) Kellerautomat KA ist ein 7-Tupel $KA = (X, Z, K, \delta, z_0, k_0, E)$ mit

X	endliches Eingabealphabet

Z	endliche Menge von Zuständen
K	endliches Kelleralphabet
$\delta : (X \cup \{\Lambda\}) \times Z \times K \rightarrow Z \times K^*$	partielle Überführungsfunktion
$z_0 \in Z$	Startzustand
$k_0 \in K$	Kellerstartzeichen
$E \subseteq Z$	nichtleere Menge von Endzuständen

Dabei muß * folgende Bedingung erfüllen:

(*) Ist für $x \in X$, $z \in Z$ und $k \in K$ $\delta(x, z, k)$ definiert, so ist $\delta(\Lambda, z, k)$ undefiniert.

Die Überführungsfunktion $\delta : (X \cup \{\Lambda\}) \times Z \times K \rightarrow Z \times K^*$ beschreibt, daß ein Tripel (x, z, k) in ein Paar (z', w) überführt wird, wobei z' der neue Zustand und w ein Wort über dem Kelleralphabet K ist, durch das das oberste Kellerzeichen zu ersetzen ist. Unter den Tripeln ist auch (Λ, z, k) zugelassen. Es soll dann das Eingabezeichen unter dem Lesekopf unberücksichtigt bleiben und der Lesekopf auf dem Eingabeband nicht verschoben werden. Damit kann der Inhalt des Kellerspeichers und der Zustand unabhängig vom Eingabezeichen verändert werden. Ganz wesentlich ist noch, daß die Überführungsfunktion nur partiell definiert ist. Ist die Funktion δ für das aktuelle Tripel (x, z, k) oder (Λ, z, k) nicht definiert, so hält der Automat an.
Die Bedingung (*) gewährleistet, daß es immer nur eine Möglichkeit gibt, die Überführungsfunktion anzuwenden.

Definition: (Konfiguration)
Es sei KA = (X, Z, K, δ, z_0, k_0, E) ein Kellerautomat. Unter einer **Konfiguration** von KA versteht man ein Tripel $(z, v, w) \in Z \times X^* \times K^*$. Dabei wird z als der aktuelle Zustand, v als das zu verarbeitende Restwort und w als der aktuelle Kellerinhalt interpretiert.

Definition: (Übergangsrelation)
Auf der Menge aller Konfigurationen definieren wir folgende Relation "$\Rightarrow$" , die den Übergang von einer Konfiguration zur nächsten in einem Schritt beschreibt:

$(z, xv, kw) \Rightarrow (z', v, w'w)$, falls $\delta(x, z, k) = (z', w')$

bzw. $(z, xv, kw) \Rightarrow (z', xv, w'w)$, falls $\delta(7, z, k) = (z', w')$

$\Rightarrow^*$ sei wieder die reflexive transitive Hülle von $\Rightarrow$, d. h. sie beschreibt den Übergang in endlich vielen Schritten bzw. das Beibehalten der Konfigurationen (vgl. mit $\Rightarrow$ und $\Rightarrow^*$ bei Grammatiken).

Definition: (akzeptiertes Wort, Sprache)
Es sei KA = (X, Z, K, δ, z_0, k_0, E) ein Kellerautomat. KA **akzeptiert** das Wort $w \in X^*$, falls

$(z_0, w, k_0) \Rightarrow^* (z, \Lambda, v)$ mit $z \in E$ gilt

Die **Sprache** L(KA) des Automaten KA ist die Menge aller Wörter, die von KA akzeptiert werden, d.h. $L(KA) = \{w \in X^* \mid (z_0, w, k_0) \Rightarrow^* (z, \Lambda, v) \text{ mit } z \in E\}$

Beispiele

1) $KA = (X, Z, K, \delta, z_0, k_0, E)$ mit

$X = \{a, b\}$

$Z = \{z_0, z_1, z_2\}$

$K = \{k_0, a\}$

$E = \{z_2\}$

und δ wie folgt:

$\delta(a, z_0, k_0) = (z_0, ak_0)$ (1)

$\delta(a, z_0, a) = (z_0, aa)$ (2)

$\delta(b, z_0, a) = (z_1, \Lambda)$ (3)

$\delta(b, z_1, a) = (z_1, \Lambda)$ (4)

$\delta(\Lambda, z_1, k_0) = (z_2, k_0)$ (5)

Wenden wir diesen Automaten zunächst auf das Eingabewort $w = a^3 b^3$ an. Dann gilt:

$(z_0, a^3 b^3, k_0) \Rightarrow$ (1 angewendet)

$(z_0, a^2 b^3, ak_0) \Rightarrow$ (2 angewendet)

$(z_0, ab^3, a^2 k_0) \Rightarrow$ (2 angewendet)

$(z_0, b^3, a^3 k_0) \Rightarrow$ (3 angewendet)

$(z_1, b^2, a^2 k_0) \Rightarrow$ (4 angewendet)

$(z_1, b, ak_0) \Rightarrow$ (4 angewendet)

$(z_1, \Lambda, k_0) \Rightarrow$ (5 angewendet)

(z_2, Λ, k_0) Stop

Es ist leicht einzusehen, daß KA alle Wörter der Form $a^n b^n$ akzeptiert. Durch (1) wird das erste a im Keller abgelegt, ebenso alle folgenden a's durch (2). Beim Auftreten eines b im Eingabewort wird durch (3) ein a vom Keller gelöscht, und der Automat geht in den Zustand z_1 über. Danach wird durch (4) für jedes weitere b im Eingabewort ein a im Keller gelöscht. Anschließend geht der Automat durch (5) in den Endzustand.

Satz:
Es sei $KA = (X, Z, K, \delta, z_0, k_0, E)$ ein (deterministischer) Kellerautomat. Dann existiert eine kontextfreie Grammatik G derart, daß $L(G) = L(KA)$ ist.

Die Umkehrung des Satzes gilt nicht. Zum Beweis kann man die Sprache

$$L = \{ vv^R \mid v, v^R \in \{0, 1\}^*, \text{ und } v^R \text{ ist die Umkehrung von } v\}$$

heranziehen, die kontextfrei ist, aber nicht durch einen (deterministischen) Kellerautomaten akzeptiert werden kann. Das hängt damit zusammen, daß der "Umkehrpunkt" nicht hervorgehoben ist, sondern erst noch durch Probieren gefunden werden muß. Abhilfe bringt hier ein nichtdeterministischer Kellerautomat, bei dem $\delta(x, z, k)$ nicht mehr aus einem einzigen Zustand besteht, sondern eine (möglicherweise auch leere) Menge von Zuständen erfaßt:

Definition: (nichtdeterministischer Kellerautomat)
Ein nichtdeterministischer Kellerautomat KA ist ein 7-Tupel $KA = (X, Z, K, \delta, z_0, k_0, E)$, wobei mit Ausnahme von δ alle Komponenten dieselbe Bedeutung haben wie bei deterministischen Automaten. Die Überführungsfunktion δ bildet die Tripel aus $X \times Z \times K$ in die Potenzmenge von $Z \times K^*$ab, also in die Menge aller Teilmengen von $Z \times K^*$.

Die Überleitungsrelation "$\Rightarrow$" und die Sprache eines nichtdeterministischen Kellerautomaten können dann wie folgt definiert werden:

$(z, xv, kw) \Rightarrow (z', v, w'w)$, falls $(z', w') \in \delta(x, z, k)$
$L(KA) = \{w \in X^* \mid (z_0, w, k_0) \Rightarrow^* (z, \Lambda, v) \text{ mit } z \in E\}$

Beispiel:
$KA = (X, Z, K, \delta, z_0, k_0, E)$ mit

$X = \{0, 1\}$
$Z = \{z_0, z_1, z_2, z_3\}$
$K = \{k_0, 0, 1\}$
$E = \{z_0, z_3\}$

und δ wie folgt:

$\delta(0, z_0, k_0) = \{(z_1, 0k_0)\}$ (1)
$\delta(1, z_0, k_0) = \{(z_1, 1k_0)\}$ (2)
$\delta(0, z_1, 1) = \{(z_1, 01)\}$ (3)
$\delta(1, z_1, 0) = \{(z_1, 10)\}$ (4)
$\delta(0, z_1, 0) = \{(z_1, 00), (z_2, \Lambda)\}$ (5)
$\delta(1, z_1, 1) = \{(z_1, 11), (z_2, \Lambda)\}$ (6)
$\delta(0, z_2, 0) = \{(z_2, \Lambda)\}$ (7)
$\delta(1, z_2, 1) = \{(z_2, \Lambda)\}$ (8)
$\delta(\Lambda, z_2, k_0) = \{(z_3, k_0)\}$ (9)

Allen Tripeln, die hier nicht auftreten wird die leere Menge zugeordnet. (10)

1. Eingabewort: 001100
$(z_0, 001100, k_0) \Rightarrow$ (1 anwenden)

$(z_1, 01100, 0k_0) \Rightarrow$ (5 anwenden)
$(z_1, 1100, 00k_0) \Rightarrow$ (4 anwenden)
$(z_1, 100, 100k_0) \Rightarrow$ (6 anwenden)
$(z_2, 00, 00k_0) \Rightarrow$ (7 anwenden)
$(z_2, 0, 0k_0) \Rightarrow$ (7 anwenden)
$(z_2, \Lambda, k_0) \Rightarrow$ (9 anwenden)
(z_3, Λ, k_0) STOP!! Also 001100 0 L(KA)

2. Eingabewort: 001
$(z_0, 001, k_0) \Rightarrow$ (1 anwenden)
$(z_1, 01, 0k_0) \Rightarrow$ (5 anwenden)
$(z_1, 1, 00k_0)$ oder $(z_2, 1, k_0) \Rightarrow$ (4 anwenden) bzw. (10 anwenden)
$(z_1, \Lambda, 100k_0) \Rightarrow$ (10 anwenden) $\Rightarrow$ STOP!! Also 001 $\notin$ L(KA).

Anmerkung: Im Gegensatz zu endlichen Automaten sind nichtdeterministische Kellerautomaten stärker als deterministische, d. h. es gibt Sprachen, die von nichtdeterministischen Kellerautomaten akzeptiert werden, aber nicht von deterministischen. Eine solche Sprache ist die zuletzt betrachtete.

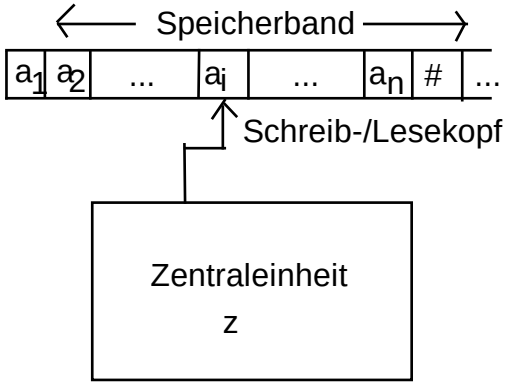
Beim Kellerautomaten besteht die Einschränkung, daß nur auf das oberste Kellersymbol zugegriffen werden kann. Es besteht aber keine Möglichkeit, darunter liegende Zeichen zu lesen oder zu ändern.
Diese Einschränkung werden wir im folgenden aufheben.

3. 2. 3. Turingmaschinen

Alan Turing entwickelte bereits 1936 das Konzept für das später nach ihm benannte Maschinenmodell, also zu einem Zeitpunkt als reale Computer noch nicht existierten. Die Turingmaschinen sind somit eine der ersten Präzisierungen des Algorithmusbegriffs gewesen, d. h. jede berechenbare Funktion kann durch eine Turingmaschine implementiert werden und umgekehrt repräsentiert jede Turingmaschine eine berechenbare Funktion. Im Hinblick auf die prinzipiellen Möglichkeiten eines Rechners ist dieses Maschinenmodell ebenso leistungsstark wie moderne Hochleistungsrechner.
Als Akzeptoren eignen sie sich zur Charakterisierung von Typ-0- und Typ-1-Sprachen.

Unter einer Turing-Maschine können wir uns ein Gerät vorstellen, das aus einer Zentraleinheit mit einem Schreib-/Lesekopf sowie einem einseitig unbegrenzten Band besteht. Die Zentraleinheit kann - ähnlich wie bei einem endlichen Automaten - endlich viele interne Zustände $z_0, z_1, z_2, \ldots z_n$ annehmen. Das Band dient als Eingabe- und Speicherband. Es ist in einzelne Felder unterteilt, die jeweils genau ein Zeichen speichern können. Der Schreib-/Lesekopf kann den Inhalt eines Feldes überschreiben bzw. lesen und auf dem Band um eine Position nach links oder rechts gerückt werden

(bzw. das Band selbst).
Da jedes Feld eines Bandes gelesen und verändert werden kann, ist das Band mit dem Hauptspeicher eines modernen Computers vergleichbar. In der Fachliteratur sagt man auch, daß die Turingmaschine durch die einseitige Unbegrenztheit des Bandes "potentiell unendlich viele" Felder hat. Das bedeutet, daß - wie bei einem realen Computer - zwar immer nur endlich viele Speicherzellen zur Verfügung stehen, aber - im Gegensatz zum realen Computer - immer so viele wie benötigt werden.



Formal kann eine Turingmaschine wie folgt definiert werden:

Definition (Turingmaschine)
Eine Turingmaschine TM ist ein 6-Tupel TM = $(X, B, Z, \delta, z_0, E)$ mit

X	endliches Eingabealphabet ($\# \notin X$)
B	endliches Bandalphabet ($X \cup \{\#\} \subseteq B$)
Z	endliche Menge von Zuständen
$\delta : Z \times B \rightarrow Z \times B \times \{L, R, N\}$	partielle Überführungsfunktion
$z_0 \in Z$	Startzustand
$E \subseteq Z$	nichtleere Menge von Endzuständen

Bei Turingmaschinen unterscheiden wir also zwischen dem eigentlichen Eingabealphabet und dem Bandalphabet. Das Bandalphabet beinhaltet alle Zeichen, die die Maschine lesen und verarbeiten kann. Es umfaßt neben dem Eingabealphabet in jedem Falle noch das Leersymbol #, das anzeigt, daß ein Feldinhalt leer, d. h. undefiniert ist.
δ wird üblicherweise durch eine Zustandstafel angegeben, die für jedes Paar (b, z) mit $x \in B$ und $z \in Z$ ein Tripel (b', z', r) mit $b' \in B$, $z' \in Z$ und $r \in \{L, R, N\}$ enthalten kann. Da δ als partielle Funktion vereinbart ist, ist an den Stellen der Tafel, wo * nicht definiert ist, kein Eintrag.
Eine Zuordnung (b, z) (b', z', r) bedeutet dabei folgendes:

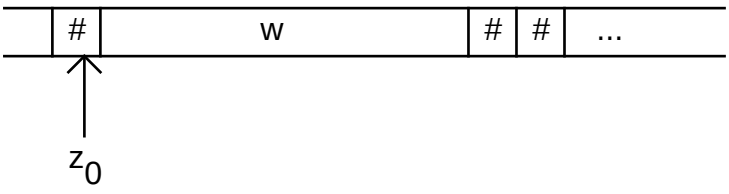
Wenn sich die Turingmaschine im Zustand z befindet und das Feld unter dem Schreib-/Lesekopf mit b beschriftet ist, dann geht sie in den Zustand z' über, ersetzt b durch b' und verschiebt den Schreib-/Lesekopf um eine Position nach links (r =L), oder um eine Position nach rechts (r=R) oder verharrt in der aktuellen Position (r=N).

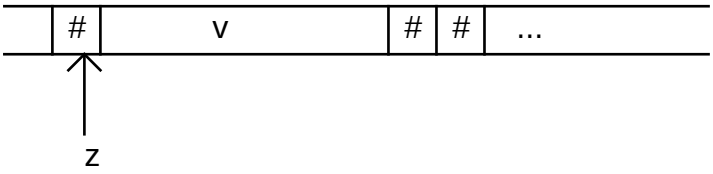
Eine präzise Beschreibung der Arbeitsweise einer Turingmaschine erfolgt - ähnlich wie schon bei den Kellerautomaten - über den Begriff der Konfiguration.
Zunächst aber wollen wir klären, was wir unter der aktuellen Bandinschrift zu verstehen haben.Sie beginnt mit dem Zeichen im ersten Feld des Bandes und endet

- im aktuellen Feld des Schreib-/Lesekopfes, fallls es rechts davon nur Leerfelder (d. h. mit Inhalt #) gibt
- ansonsten im am weitesten rechts liegenden Feld, das kein Leerfeld ist.

Definition (Konfiguration einer Turingmaschine)
Es sei TM = (X, B, Z, δ ,z_0, E) eine Turingmaschine. Eine Konfiguration von TM ist ein 4-Tupel (v, b , w, z) $\in$ B* x B x B* x Z . Dabei ist vbw als die aktuelle Bandinschrift und z als der aktuelle Zustand zu interpretieren. Der Schreib-/Lesekopf befindet sich auf dem Zeichen b.
Ist w $\in$ X* ein Eingabewort, dann nennen wir (Λ, #, w, z_0) die **Initialkonfiguration** bzgl. w und (Λ, #, v, z) mit v $\in$ B* eine **Finalkonfiguration**, falls es für (Λ, #, v, z) keine Folgekonfiguration gibt und z $\in$ E ist.

Bildlich können wir uns Initial- bzw. Finalkonfiguration wie folgt darstellen:





Eine Turingmaschine stoppt, wenn δ(b, z) nicht definiert ist.

Definition:
Ein Wort **w** wird von einer Turingmaschine TM = (X, B, Z, δ, z_0, E) **akzeptiert**, wenn die Maschine mit der Initialkonfiguration bzgl. **w** beginnend nach endlich vielen Schritten in einer

Finalkonfiguration stoppt.
Die **Sprache** L(TM) einer Turingmaschine ist die Menge aller Wörter, die von TM akzeptiert werden.

Beispiele:

1) $L = \{a^n b^n c^n \mid n \in N\}$

Diese Sprache wird durch die folgende TM $= (X, B, Z, \delta, z_0, E)$ akzeptiert:

$X = \{a, b, c\}$

$B = \{a, b, c, 0, 1, 2, \#\}$

$Z = \{\{z_0, z_1, ..., z_6\}$

$E = \{z_6\}$ und δ gemäß der folgenden Tabelle definiert:

	a	b	c	0	1	2	#
z_0	-	-	-	-	-	-	$(z_1, \#, R)$
z_1	$(z_2, 0, R)$	-	-	-	$(z_5, 1, R)$	-	-
z_2	(z_2, a, R)	$(z_3, 1, R)$	-	-	$(z_2, 1, R)$	-	-
z_3	-	(z_3, b, R)	$(z_4, 2, L)$	-	-	$(z_3, 2, R)$	-
z_4	(z_4, a, L)	(z_4, b, L)	-	$(z_1, 0, R)$	$(z_4, 1, L)$	$(z_4, 2, L)$	-
z_5	-	-	-	-	$(z_5, 1, R)$	$(z_5, 2, R$	$(z_6, \#, L)$
z_6	-	-	-	$(z_6, 0, L)$	$(z_6, 1, L)$	$(z_6, 2, L)$	-

Betrachten wir die Konfigurationsfolge bei Anwendung der TM auf das Wort w = abc:

$(\Lambda, \#, abc, z_0) \Rightarrow (\#, a, bc, z_1) \Rightarrow (\#0, b, c, z_2) \Rightarrow (\#01, c, \Lambda, z_3) \Rightarrow$

$(\#0, 1, 2, z_4) \Rightarrow (\#, 0, 12, z_4) \Rightarrow (\#0, 1, 2, z_1) \Rightarrow (\#01, 2, \Lambda, z_5) \Rightarrow (\#012, \#, \Lambda, z_5) \Rightarrow$

$(\#01, 2, \Lambda, z_6) \Rightarrow (\#0, 1, 2, z_6) \Rightarrow (\#, 0, 12, z_6) \Rightarrow (\Lambda, \#, 012, z_6)$ STOP!!

Bei Anwendung der TM auf das Wort w = ab erhalten wir:

$(\Lambda, \#, ab, z_0) \Rightarrow (\#, a, b, z_1) \Rightarrow (\#0, b, \Lambda, z_2) \Rightarrow (\#01, \#, \Lambda, z_3)$ STOP, denn $\delta(\#, z_3)$ ist nicht definiert. Da $z_3 \notin E$ ist, wird w = ab nicht von der TM akzeptiert.

2) $L = \{w \in \{a, b\}^* \mid$ w enthält mindestens zwei a's$\}$

Diese Sprache wird durch die folgende TM $= (X, B, Z, \delta, z_0, E)$ akzeptiert:

$X = \{a, b\}$

$B = \{a, b, \#\}$

$Z = \{z_0, z_1, z_2, z_3\}$

$E = \{z_3\}$ und δ gemäß folgender Zustandstafel spezifiziert:

	a	b	#
z_0	-	-	(z_1, #, R)
z_1	(z_2, a, R)	(z_1, b, R)	-
z_2	(z_3, a, L)	(z_2, b, R)	-
z_3	(z_3, a, L)	(z_3, b, L)	-

Es wird offensichtlich kein Zeichen verändert, d. h. der Automat hat nur lesende Funktion und entspricht in seiner Wirkung einem endlichen Automaten. Analog kann gezeigt werden, daß auch jeder Kellerautomat durch eine Turingmaschine simuliert werden kann.
Daß die Turingmaschinen aber deutlich mehr leisten kommt auch in dem folgenden Satz zum Ausdruck.

Satz
Zu jeder Typ-0-Grammatik G gibt es eine Turingmaschine TM mit L(G) = L(TM) und umgekehrt.

Damit haben wir eine exakte Charakterisierung der Typ-0-Sprachen durch einen Akzeptor gefunden.

Wir haben einleitend schon darauf hingewiesen, daß Turingmaschinen eine der ersten Präzisierungen des Berechenbarkeitsbegriffes waren. Dieser Thematik wollen wir uns abschließend noch kurz zuwenden.

Definiton (Turing-Berechenbarkeit)
Es sei TM = (X, B, Z, δ, z_0, E) eine Turingmaschine und M_1, M_2 f X*.
Wir sagen dann: TM berechnet die Funktion f : $M_1 \rightarrow M_2$, wenn folgendes gilt:

(1) für alle w $\in$ M_1 gilt: (Λ, #, w, z_0) geht nach endlich vielen Schritten in (Λ, #, f(w), z) über, wobei (Λ, #, f(w), z) eine Finalkonfiguration ist.

(2) für w $\notin$ M_1 geht die Maschine nie in eine Finalkonfiguration über, d. h. das Verhalten der Maschine ist unbestimmt.

Eine Funktion **f** heißt **Turing-berechenbar**, wenn es eine Turingmaschine gibt, die f berechnet.

Beispiele:

1) Die Funktion f(n) = n + 1 ist Turing-berechenbar
Wir stellen die natürlichen Zahlen unär dar, d. h. wir benutzen das Alphabet {|} und stellen die Zahl n durch n Striche dar ("Bierdeckel-Notation")
Die entsprechende TM muß dann nur an das Ende der codierten Zahl eine 1 anhängen.

2) Die Funktion f(x, y) = x + y ist Turing-berechenbar.
x und y werden wiederum unär dargestellt. Die Initialkonfiguration sieht dann wie folgt aus: (Λ, #, x#y, z_0). D. h. die Argumente x und y werden durch ein Leersymbol getrennt. Die Arbeitsweise der TM besteht dann darin, daß dieses trennende Leersymbol gelöscht wird.
Diese Vorgehensweise kann auf beliebige k-stellige Funktionen erweitert werden.

3. 2. 4. Linear beschränkte Automaten

Unter einem linear beschränkten Automaten verstehen wir eine Turingmaschine bei der nur ein beschränkter Teil des Bandes benutzt werden darf. Dazu wird ein Begrenzungssymbol $ eingeführt, das vom Schreib-/Lesekopf nicht überschritten werden darf, d. h. der Schreib-/Lesekopf darf sich nur zwischen dem linken Bandende und dem Begrenzungssymbol bewegen.
Es wird nicht gefordert, daß diese Bandbeschränkung für jede Berechnung gleich ist (d. h. daß das Begrenzungssymbol immer an derselben Position steht), sondern die Länge des verfügbaren Bandabschnittes soll linear von der Länge des jeweiligen Eingabewortes abhängen.
Die Leistungsfähigkeit dieses Maschinenmodells liegt zwischen der von Kellerautomaten und Turingmaschinen.

Definition (linear beschränkter Automat)
Ein linear beschränkter Automat ist eine nichtdeterministische Turingmaschine (X, B, Z, δ, z_0, E) mit X, B, Z, z_0 und E wie in der Definition der Turingmaschine. Zusätzlich gilt $\$ \in B$, und δ ist eine totale Funktion von B x Z in die Potenzmenge von B x Z x {L, R, N}. Es wird gefordert, daß auf dem Band ein Begrenzungssysmbol $ steht und daß sich der Schreib-/Lesekopf nur zwischen dem linken Bandende und dem Begrenzungsymbol hin- und herbewegen kann. In Abhängigkeit vom jeweiligen Eingewort $w \in X^*$ soll die verfügbare Bandlänge durch eine Funktion $l_B : X^* \rightarrow N$ mit $l_B(w) := k \cdot |w| + c$ (mit $k, c \in N$) beschränkt sein.

Im Prinzip arbeitet ein linear beschränkter Automat wie eine nichtdeterministische Turingmaschine. Der nachfolgende Satz stellt eine Beziehung zu Typ-1-Sprachen her.

Satz:
Zu jeder Typ-1-Grammatik G gibt es einen linear beschränkten Automaten LBA mit L(G) = L(LBA) und umgekehrt.

Damit wären alle Sprachklassen sowohl durch akzeptierende Automaten als auch durch erzeugende Grammatiken charakterisiert.

4. Literatur

Brecht, W.: Theoretische Informatik - Grundlagen und praktische Anwendungen. Friedr. Vieweg &

Sohn Verlagsgesellschaft mbH, Braunschweig/Wiesbaden 1995
Breier, N.:Ein Plädoyer für die Registermaschine.- In: LOG IN 9 (1989) H. 1, S. 29-32
Breier, N.:Grenzen des Computereinsatzes (Teil 1). - In: LOG IN 10 (1990) H. 1, S. 42-50
Breier, N.:Grenzen des Computereinsatzes (Teil 2). - In: LOG IN 10 (1990) H. 3, S. 37-42
Breier, N.:Grenzen des Computereinsatzes. Handreichung zum Informatikunterricht, Freie und Hansestadt Hamburg, Behörde für Schule, Jugend und Berufsbildung, Amt für Schule, Referat S 13/12, Hamburg 1991
Breier, N.:Berechenbarkeit und Entscheidbarkeit. - In: LOG IN 11 (1991) H. 3, S. 29-35
Breier, N.:Algorithmisch lösbare und unlösbare Probleme. In: RAABITs Informatik, Raabe Schulbuchverlag, Heidelberg 1995
Gasper, F.; Leiß, l.; Spengler, M.; Stimm, H.: Technische und theoretische Informatik. Bayrischer Schulbuch-Verlag, München 1992
Hopcroft, J. E.; Ullman, J. D.: Formal languages and their relation to Automata. Addison-Wesley Publishing Company 1969
Malcev, A. I.: Algorithmen und rekursive Funktionen. Akademie-Verlag, Berlin 1974
Schöning, U.: Theoretische Informatik kurz gefaßt. BI-Wissenschaftsverlag Mannheim/Leipzig/ Wien/Zürich 1992
Sander, P.; Stucky, W.; Herschel, R.: Automaten, Sprachen, Berechenbarkeit. B. G. Teubner, Stuttgart 1992