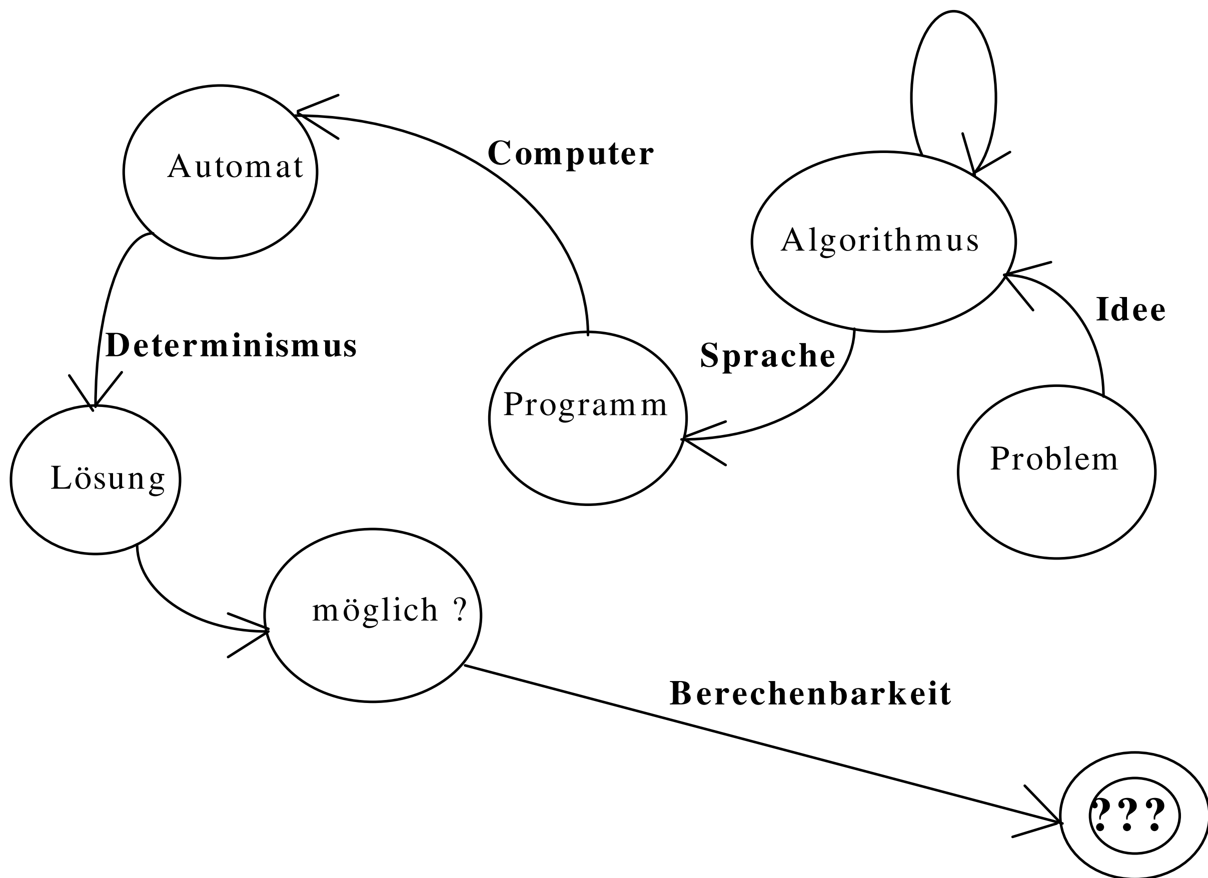
Theoretische Informatik

Planung eines Kurses in der Jahrgangsstufe 13 I



Entworfen im Rahmen eines Wochenlehrgangs „Didaktik der Informatik“
von

Günter Battenfeld
Petra Grimm
Dr. Jürgen Poloczek
Dr. Elke Schmidt
Rudolf Waha

Weilburg, im Juni 1996

Vorbemerkungen

In den neuen Kursstrukturplänen wird für das Fach Informatik unter den verbindlichen Inhaltsbereichen als Bereich II „Universelle symbolverarbeitende Maschine - Mensch und Maschine" genannt. „In diesem Inhaltsbereich stehen die theoretischen und technischen Grundlagen der maschinellen Informationsverarbeitung sowie ihre prinzipiellen Möglichkeiten und Grenzen im Vordergrund der Betrachtung" ([KSP 94], S. 13). Zur Konkretisierung wird in dem in den KSP vorgeschlagenen Kursdurchgang für die Jahrgangsstufe 13 I ein Kurs mit dem Thema „Grundlagen der theoretischen Informatik" vorgeschlagen.

Da für ein solches Kursthema bisher nur wenig Unterrichtserfahrung vorliegt, hatte es sich unsere Gruppe zum Ziel gesetzt, hierfür einen Vorschlag auszuarbeiten, wie ein solcher Kurs realisiert werden könnte. Zum Einen soll anhand einer Grobplanung dargelegt werden, wie man die zeitliche Verteilung der zu behandelnden Themen in der in einem Halbjahreskurs zur Verfügung stehenden Zeit von ca. 17 Wochen (u.U abzüglich der Zeit für Klausuren, deren Besprechung und sonstigem evtl. Unterrichtsausfall) vornehmen kann, zum Anderen soll beispielhaft für einzelne Unterrichtseinheiten und auch Unterrichtsstunden eine Planung vorgenommen werden. Der Kurs wurde jeweils für Doppelstunden geplant (zweistündiger Grundkurs). Sollte eine weitere Einzelstunde pro Woche zur Verfügung stehen, kann hier das Thema entsprechend vertieft oder Doppelstunden aufgeteilt werden.

Grundsätzliche Überlegungen

Die Themenfolge wurde für einen Grundkurs Informatik geplant. Hier sollte unserer Meinung nach zwar ein Überblick über die Gebiete der theoretischen Informatik gegeben werden, jedoch auf eine möglichst anschauliche und behutsame Weise, wobei vertiefte mathematische Methoden und Beweise vermieden werden können (siehe [KSP 94] S. 22).

Auf eine programmtechnische Realisierung der vorgestellten Probleme in der bisher behandelten Programmiersprache kann verzichtet werden. Für die Vermittlung der vorgestellten Lernziele ist dies nicht notwendig, die Ideen können eher besser ohne den Ballast der möglichen Programmierung erörtert werden. Dies erspart auch den dazu nötigen Zeitbedarf. Für die Simulation von Automaten (sowohl deterministische als auch nichtdeterministische) steht eine Simulationsumgebung zur Verfügung. Das Abbild des Automaten mit seinen Zuständen und Übergängen kann unmittelbar in das Computermodell eingegeben werden. Die Dateien für diese Simulationsumgebung sowie auch die anderen hier erwähnten Programme lassen sich von dem ftp-Server der Uni Frankfurt in ***ftp://ftp.rz.uni-frankfurt.de/pub/lwb/*** herunterladen.

Aus den verbindlichen Inhalten des Themenbereichs *Fachwissenschaft Informatik* (siehe [KSP 94] S.14) wird im ersten Teil des Kurses der Begriff der Berechenbarkeit in den Mittelpunkt gestellt. Hier bietet es sich an, auf die Fragen „Was macht eine Maschine aus, was kann ein Computer prinzipiell berechnen, wo liegen die Grenzen des Computers?" einzugehen. Dadurch gewinnen die Schüler „Einsicht in Möglichkeiten und Grenzen maschineller Informationsverarbeitung" als erzieherische Zielsetzung dieser Unterrichtseinheit (siehe [KSP 94] S. 14). Die Frage nach der Berechenbarkeit großer Systeme und ihrer gesellschaftliche Kontrolle muss in diesem Inhaltsbereich diskutiert werden (siehe [KSP 94] S. 15).

Die zweite Unterrichtseinheit befaßt sich als Schwerpunktthema mit den Automatenmodellen. Der Automat dient dabei als mathematisches Modell zur Beschreibung und Präzisierung von Datenverarbeitung (siehe [KSP 94] S. 22). Insbesondere soll auf die Modellierung von Suchmaschinen und Porgrammiersprachen durch endliche Automaten eingegangen werden. In dieser Phase kann intensiv mit dem Simulationsprogramm für Autom aten gearbeitet werden.

Fast übergangslos schließt sich hieran die dritte Einheit zum Thema Sprachen und Grammatiken an. Als Erweiterung des Automaten werden hier der Kellerautomat und die Turingmaschine eingeführt.

Auf die konkrete technische Realisierung von Automaten mit logischen Schaltungen ist in dem hier dargestellten Kurs verzichtet worden. Einerseits ist die zur Verfügung stehende Zeit sehr knapp, zum anderen läßt sich dieses Thema in dem hier dargestellten Kursablauf an keiner Stelle sinnvoll integrieren. Möglich wäre es, dies zum Thema von Hausarbeiten als Ersatz für eine zweite Klausur zu machen oder, falls genügend Zeit ist, Hardwarerealisierung am Ende des Kurses zu behandeln oder in einer Projektphase parallel zur Automatenprogrammierung in eine Schülergruppe zu verlagern.

Didaktische Überlegungen

Die theoretische Informatik hat durch die neuen Kursstrukturpläne für das Fach Informatik eine deutliche Aufwertung erfahren, und gegenüber früher wird ihre Behandlung in der Jahrgangsstufe 13 auch einen deutlich größeren Zeitraum umfassen, wenn sie denn früher überhaupt Unterrichtsgegenstand war. Anhand einiger Stichworte soll darauf eingegangen werden, welche Bedeutung die Theoretische Informatik innerhalb des Faches Informatik und auch in ihrer Auswirkung auf einen gesamtgesellschaftlichen Zusammenhang hat. Diese Stichworte entsprechen zum Teil auch den in den KSP genannten Schwerpunkten, in denen der Beitrag des Informatikunterrichts zur Allgemeinbildung der Schüler liegt. ([KSP 94] , S. 5)

Weltsicht, Beschreibung und Modellierung komplexer Systeme
Während beim Erstellen von Programmen die Modellierung der Umwelt durch Daten und Algorithmen vorgenommen wird, erforscht die Theoretische Informatik die Modellierung der Vorgehensweise von Informationsverarbeitung an sich. Sie hat daher eine viel grundlegendere Bedeutung und ist in ihrer Sichtweise auch nicht so vielen zeitlichen Änderungen unterworfen, als dies bei den Programmiersprachen als Unterrichtsgegenstand der Fall ist.
„Nicht Kenntnis technischer Einzelheiten, sondern Wissen um allgemeine Zusammenhänge hat Zukunftsbedeutung. .... Was heute aus Sicht der Praxis bzw. der Technik der neueste Schrei ist, kann morgen bereits überholt sein; es wäre daher höchst fatal, wenn der Informatikunterricht vergängliches Detailwissen vermitteln wollte. Was bleibt, sind die grundlegenden Ideen: um sie muß es im Unterricht daher gehen. Ideen kann aber nur die Theorie vermitteln.“([Bau93] S. 6).
Insofern kann auch die Theoretische Informatik ihren Beitrag dazu leisten, „Schülerinnen und Schüler mit ihrer Welt vertraut zu machen und ihnen den Aufbau eines zeitgemäßen Weltbildes zu ermöglichen.“ ([KSP 94] , S. 5)

Technikgläubigkeit, Grenzen des Machbaren, verantwortungsbewußter Umgang

Der Computer ist in der heutigen Welt allgegenwärtig. Seinen Möglichkeiten scheinen kaum Grenzen gesetzt. Auch in seinem bisherigen Unterricht hat der Schüler erfahren, daß er unter Verwendung von geeigneten Algorithmen immer komplexere Probleme lösen konnte. In diesem Kurs kann den Schülern die Einsicht vermittelt werden, daß die Möglichkeiten, die der Computer zur Lösung von Problemen bietet, nicht nur durch die jeweilige technische Ausführung und die mangelnde Effizienz der verwendeten Algorithmen begrenzt sind. Vielmehr stößt er hier an die prinzipiellen Grenzen von formalen Systemen.
Baumann sagt über Unlösbarkeit von Problemen „Und zwar ist ihre Lösung aus prinzipiellen Gründen unmöglich, das heißt, daß sich diese Unmöglichkeit durch menschliche Erfindungskraft, wissenschaftlichen oder technischen Fortschritt ... niemals wird aufheben lassen. Es handelt sich damit um Grenzen nicht so sehr eines Geräts (des Computers) als des menschlichen Denkens und Tuns selbst." ([Bau93] S. 246)
Aus dieser Erkenntnis heraus wird der Schüler in die Lage versetzt, die Möglichkeiten, die der Computer und die Informatik als Wissenschaft überhaupt bietet, um Probleme aus der real existierenden Welt zu lösen, realistisch abzuschätzen.

Zusammenhang Theorie-Praxis

In diesem Kurs erfährt der Schüler, welche theoretischen Grundlagen den Werkzeugen zugrunde liegen, die er im bisherigen Unterricht zur Lösung von Problemen angewandt hat. Er kann an Beispielen erkennen, wie diese Grundlagen zur Entwicklung von Anwenderprogrammen (z.B. Programmiersprachen) herangezogen werden können. Die Korrektheit und die Effizienz praktischer Algorithmen werden überprüfbar. Zudem bieten sich immer wieder Möglichkeiten, die Modelle, die erarbeitet werden, auch unmittelbar auf die reale Umwelt zu übertragen.

„Im Bereich der theoretischen Informatik werden darüber hinaus Beschreibungsformen gewählt, die sehr weitgehend in der praktischen Informatik Verwendung finden. .... Wir haben damit wieder die beiden Seiten der Informatik eng beieinander, die das Fach für die Schüler so attraktiv machen ... „ ([Mod92] , S. 352)

Unterrichtliche Voraussetzungen

Im Prinzip müssen an diesen Kurs keine unterrichtlichen Voraussetzungen geknüpft werden. Er eignet sich daher auch, wenn in diesem Kurs Schüler sein sollten, die als Seiteneinsteiger erst in Jahrgangsstufe 13 I mit Informatik beginnen, weil sie etwa noch Unterrichtsverpflichtungen im 3. Aufgabenfeld erfüllen müssen.

Gleichwohl muß gesagt werden, daß es an einigen Stellen des Kurses von Vorteil ist, wenn die Schüler Erfahrung im Lösen von Problemen mit Hilfe einer Programmiersprache haben. Erst dann erschließt sich ihnen wirklich die Bedeutung der Grenzen der Machbarkeit, die in diesem Kurs zutage treten. Die einerseits vorhandenen Unterschiede zwischen theoretischer und praktischer Informatik werden für diese Schüler deutlicher, während andererseits die doch enge Verbindung zwischen der Theorie (z.B. der Automaten) und den Möglichkeiten und der Korrektheit einer Problemlösung mittels einer Programmiersprache eher erkennbar wird.

Grobplanung des Kurses

- Berechenbarkeit - Entscheidbarkeit Determinismus
  geschätzter Zeitaufwand: 4 Doppelstunden
- Automaten
  geschätzter Zeitaufwand: 6 Doppelstunden
- Sprachen und Grammatiken
  geschätzter Zeitaufwand: 6 Doppelstunden
- Weitere Themen aus den unter „Weitere Inhalte“ aufgeführten je nach zur Verfügung stehenden Zeit

Planung der einzelnen Unterrichtseinheiten

1. Unterrichtseinheit: Berechenbarkeit - Entscheidbarkeit - Determinismus

Lernziele

Die Schüler sollen

- die Laufzeit von Programmen an Beispielen abschätzen können
- den Begriff der Berechenbarkeit kennenlernen
- erkennen, daß Algorithmen mit exponentiellem Zeitaufwand für die Praxis nur bedingt geeignet sind
- wissen, daß informatische Probleme unterschiedliches Verhalten bezüglich ihrer algorithmischen Lösbarkeit haben
- erkennen, daß es auch Probleme gibt, die prinzipiell zwar berechenbar, aber praktisch nicht durchführbar sind
- lernen, daß es auch Probleme gibt, die prinzipiell nicht berechenbar sind
- lernen, daß es kein allgemeines Verfahren gibt, um zu testen, ob ein beliebiges Programm bei einer Eingabe überhaupt anhält
- lernen, daß es kein allgemeines Verfahren gibt, um zu testen, ob zwei Programme äquivalent sind
- das Halteproblem und Äquivalenzproblem als ein nachweislich unentscheidbares Problem kennenlernen.

Detailliertere Planung der Unterrichtseinheit Berechenbarkeit

1./2. Stunde:

Ziele: Begriff der exponentiellen Laufzeit
Einführung der Sprechweise: berechenbar, aber nicht durchführbar

Einführungsbeipiel soll ein (prinzipiell) berechenbares, (praktisch) aber nicht durchführbares Problem sein. Es sollte folgenden Anforderungen genügen:

- durch systematisches Probieren soll eine Lösung gefunden werden können, deren Korrektheit unmittelbar einsichtig ist

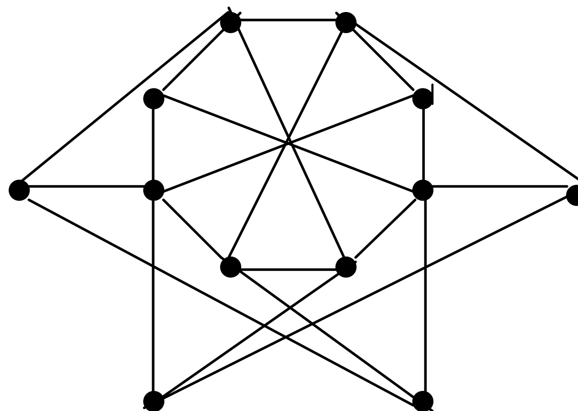
- elementare Überlegungen zeigen rasch, daß ein „naiver“ Algorithmus zur Auffindung einer Lösung exponentielles Laufzeitverhalten hat

Geeignet erscheint das Beispiel „Ausschußsitzungstermine von Parlamentsabgeordneten“. Es wird im Hinblick auf die später gemeinsam durchzuführende Laufzeitbetrachtung zunächst allgemein formuliert, den Schülern daraufhin in einer Konkretisierung (für 12 Ausschüsse und 3 Farben) als Färbungsaufgabe gestellt, um sie mit dem Problem vertraut werden zu lassen.

Aufgabe:
Die Abgeordneten eines Parlaments gehören n verschiedenen Ausschüssen an. Jeder Ausschuß tagt jede Woche genau einmal. Ist ein Abgeordneter Mitglied in zwei verschiedenen Ausschüssen, so dürfen diese nicht zur gleichen Zeit stattfinden. Wir möchten wissen, ob wir mit k verschiedenen Sitzungsterminen auskommen.

Erfassen wir die Aufgabenstellung durch einen Graphen, in dem jeder Ausschuß durch einen Knoten dargestellt wird und zwei Knoten genau dann durch eine Kante verbunden werden, wenn es einen Abgeordneten gibt, der in beiden Ausschüssen vertreten ist! Das Problem der Sitzungstermine läßt sich als Färbungsproblem des Graphen auffassen. Läßt sich jedem Knoten des Graphen eine von k Farben so zuordnen, daß zwei durch eine Kante verbundene Knoten unterschiedliche Farben besitzen?

Löse dieses Färbungsproblem für den folgenden Graphen mit 12 Ausschüssen (Knoten):



Nachdem ein Schüler eine von ihm gefundene Lösung vorgestellt hat, betrachten wir einen *naiven Algorithmus* zur Lösung des allgemeinen Färbungsproblems:

Bei k zur Verfügung stehenden Farben und n Knoten führen k Färbungsmöglichkeiten pro Knoten auf k^n Färbungsmöglichkeiten des Graphen. Ein Graph mit n Knoten hat „2 aus n“ = n(n-1)/2 Knotenpaare. Wir untersuchen jedes Knotenpaar und testen ihre beiden Farben, sofern das Paar durch eine Kante verbunden ist. Hierzu benötigen wir maximal $n(n-1)/2 * k^n$ Schritte.

Auf eine Vereinfachung dieses Terms durch eine weitere Abschätzung nach oben verzichten wir, denn der Term $n^2 * k^n$ würde in der folgenden Aufgabe die von unserem Ziel ablenkende

Frage aufwerfen, wie weit die hierfür konkret berechneten Laufzeiten von denen für $n(n-1)/2 * k^n$ abweichen.

Aufgabe:
Für k=3 ergibt sich für die maximale Anzahl von Schritten der Term $T(n) = n(n-1)/2 * 3^n$. Berechne die sich hieraus ergebenden Laufzeiten für folgende Knotenzahlen n: 12, 20, 30, 50, 100. Nimm hierbei an, daß der zugrunde liegende Computer 100000 Schritte pro Sekunde ausführen kann.
Lösung:

Eingabegröße *n*	12	20	30	50	100
Laufzeit *T*	350,8s	76,7d	28400a	$2{,}79 \cdot 10^{14}$a	$8{,}09 \cdot 10^{38}$a

Warum wächst die Laufzeit so schnell?
Mit der folgenden, nicht eigenhändig berechneten Tabelle ließe sich die Hürde zwischen Polynomial- und Exponentialzeit vor Augen führen:

T(n)	n				
	20	30	40	50	100
n	0,0002s	0,0003s	0,0004s	0,0005s	0,001s
n^2	0,004s	0,009s	0,016s	0,025s	0,1s
n^5	32s	243s	1024s	3125s	100000s
2^n	10s	3h	4 Monate	360a	$4 \cdot 10^{17}$a

Interessant für den Lehrer ist auch folgende Überlegung:

Nimmt man einen Rechner, der um den Faktor 1000 schneller arbeitet, so ändert sich nichts Wesentliches an der Problematik, die untenstehende Tabelle zeigt, wie der Umfang einer etwa in einer Stunde lösbaren Aufgabe wächst:

Zeitkomplexität	Problemumfang bei gegenwärtiger Rechenanlage	Problemumfang bei 1000mal schnellerer Rechenanlage
$O(n^2)$	d_1	$31{,}6 \cdot d_1$
$O(2^n)$	d_2	$d_2 + 9{,}97$
$O(3^n)$	d_3	$d_3 + 6{,}29$

Der naive Algorithmus des betrachteten Beispiels hat exponentielle Laufzeit. Algorithmen mit exponentieller Laufzeit sind (schon für relativ kleine Eingabelängen n) praktisch undurchführbar. Erinnern wir uns an Sortieralgorithmen mit polynomialer Laufzeit (z.B. Sortieren durch Auswahl: Laufzeit „verhält sich wie“ n^2) und fassen zusammen: Hinsichtlich der Laufzeit unterscheiden wir bei Algorithmen bislang:

berechenbar und durchführbar (polynomiale Laufzeit)	**berechenbar, aber nicht durchführbar (exponentielle Laufzeit)**

Die folgende Hausaufgabe soll den Schülern Gelegenheit geben, nach erfolgreichem, wenn auch leichtem Spiel bei der Suche nach einer konkreten Lösung, selbst eine Laufzeitbetrachtung zu versuchen, die in einem anderen Problemzusammenhang einzig auf die Frage zurückführt, wie viele Möglichkeiten es gibt, n Elemente einer Menge auf zwei ihrer Teilmengen zu verteilen.

Hausaufgabe:
Ein Mann hat seinen Reichtum in n Goldklumpen angelegt, die die Gewichte g_1, g_2, , ..., g_n besitzen. Nach seinem Tode soll der Reichtum zu gleichen Teilen an seine beiden Kinder fallen. Im Testament wurde jedoch festgelegt, daß kein Goldklumpen zerschlagen werden darf. Gelingt die exakte Aufteilung nicht, fällt das Vermögen an die Kirche. Gibt es eine Aufteilung der Menge $\{g_1, g_2, , g_n\}$ in zwei Mengen, so daß die Summen der Zahlen jeder Teilmenge gleich sind?

Betrachte den folgenden konkreten Fall (Gewichte g_i in Gramm):
$n = 8$, $g_1 = 10$, $g_2 = 14$, $g_3 = 19$, $g_4 = 30$, $g_5 = 38$, $g_6 = 40$, $g_7 = 45$, $g_8 = 56$
a) Suche eine Lösung für den konkreten Fall!
b) Bestimme allgemein die Laufzeit eines naiven Algorithmus, der alle Möglichkeiten der Aufteilung von n Gewichten auf die beiden Erbteilmengen nach einer Lösung durchsucht!

3./4. Stunde

Ziele: Festigung des Begriffs der exponentiellen Laufzeit

Mit der Besprechung der Hausaufgabe knüpfen wir an das Ergebnis der letzten Stunde an. Es gibt prinzipiell lösbare, praktisch jedoch undurchführbare Probleme. Darüber hinaus stellen wir die Leitfrage: Gibt es Probleme, die prinzipiell nicht berechenbar sind?

Das Beispiel „Wundersame Zahlen“ dient der Vorbereitung einer Definition des Begriffs Problem und der Hinführung zur Sprechweise: „Das Problem ist berechenbar.“ Von Anfang an soll die Formulierung eines Problems als (wohldefinierte) Funktion $f: \mathrm{IN} \rightarrow \mathrm{IN}$ abgegrenzt werden von der Frage nach der Berechenbarkeit des Problems, sprich der der Funktionswerte.

Nähern wir uns einer Beantwortung der Leitfrage durch folgendes Beispiel: Wundersame Zahlen (3n+1 - Problem) Starte mit einer beliebigen natürlichen Zahl und bestimme die Folgezahlen jeweils wie folgt: Ist eine Zahl ungerade, so ist das nächste Folgenglied das Dreifache dieser Zahl, erhöht um 1. Ist eine Zahl gerade, so ist das nächste Folgenglied die Hälfte dieser Zahl. Die Startzahl heißt wundersam, falls die Folge irgendwann auf den Wert 1 stößt.
a) Finde einige wundersame Zahlen! (Bleistift und Papier)
b) Teste mit Hilfe des folgenden Programms einige Zahlen auf Wundersamkeit:

```
PROGRAM Wundersam;
  VAR
    Startwert,n : INTEGER;
  BEGIN
    WRITE('Gib den Startwert ein: ');
    READLN(Startwert);
    WRITELN;
```

```
    n:=Startwert;
    REPEAT
      IF (n MOD 2 = 0) THEN
        n:=n DIV 2
      ELSE
        n:=3*n+1
    UNTIL n=1;
    WRITELN('Die Zahl ',Startwert,' ist wundersam.')
END.
```

Nach der Übungsphase wird die Reaktion des Programms auf Eingabezahlen diskutiert. Herauszuarbeiten ist:

- Das Programm ist prinzipiell in der Lage, eine wundersame Zahl als solche zu erkennen
- Welche Ursachen kann eine „nicht enden wollende Laufzeit“ haben?
- die Eingabezahl könnte wundersam sein, aber die Bestätigung ist praktisch undurchführbar (s.o.)
- die Eingabezahl könnte nicht wundersam sein. In diesem Fall gerät das Programm in eine Endlosschleife.

5./6. Stunde

Ziele: Definition eines Problems als Funktion f:IN →IN
Abgrenzung von der Frage der Berechenbarkeit der Funktionswerte
Erkenntnis, daß die Menge aller Programme eine Teilmenge von IN ist

Die Thematik ist theoretisch. Die Ausführungen bis zu der am Ende gestellten Hausaufgabe sind als roter Faden zu lesen, den der Lehrer sich vor der Stunde zurechtgelegt haben könnte. In Anknüpfung an das Ergebnis der Diskussion des Beispiels ‘Wundersame Zahlen’ gelangen wir wie folgt zu einer Definition des Begriffs ‘Problem’. Die Problemstellung des letzten Beipiels läßt sich als Zuordnund auffassen: einer natürlichen Zahl (Eingabe) wird der Wert 1 (ja) zugeordnet, falls sie wundersam ist, andernfalls den Wert 0 (nein), d.h. das Problem ist eine Funktion f:IN →IN, mit

f(n) = 1, falls die Folge mit Startwert n das Element 1 enthält
0, falls dies nicht der Fall ist.

Davon zu trennen ist die Frage, ob die Funktionswerte berechenbar sind. Im vorliegenden Beispiel liegt ein Algorithmus vor, der wundersame Zahlen prinzipiell erkennt, also f(n)=1 berechnen kann, bei nicht wundersamen Zahlen jedoch nicht imstande ist, den zugehörigen Funktionswert 0 zu ermitteln. Die Frage, ob es einen anderen, auch für nicht-wundersame Zahlen terminierenden Algorithmus gibt, bleibt offen und ist bis heute unbeantwortet. In dem obigen Beispiel ordnet das Problem f einer eingegebenen natürlichen Zahl $n \in$ IN eine natürliche Zahl aus $\{0,1\}$ als Ausgabe zu. Allgemeiner:

Was eine Eingabe auch immer ist (Texte, Zahlen, Bilder...), auf der untersten formalen Ebene ist es eine Folge aus 0 und 1. Betrachtet man diese Folge als Dualzahl, dann ist eine Eingabe eines Zahl $x \in$ N. Ebenso ist eine Ausgabe eine Zahl $y \in$ IN. Demnach definieren wir:
Ein Problem ist eine Funktion f: IN →IN.

Nach der Darstellung eines Problems als Funktion f stellt sich die Frage: Wie berechnen wir die Funktionswerte?

Wir benötigen eine eindeutige, endliche Folge von Anweisungen, einen Algorithmus. Ein solcher Algorithmus hat eine endliche Beschreibung über einem Alphabet S, etwa dem aus Buchstaben, Ziffern und einigen Sonderzeichen. Jeder Algorithmus ist eine endliche Zeichenkette mit Zeichen aus S und läßt sich durch ein Programm darstellen. Ein Programm ist, wie immer es auch formuliert oder niedergeschrieben wird, letztlich auch als Folge von 0 oder 1 anzusehen, d.h. auch als Dualzahl zu interpretierbar.

Kurz: Ein Programm ist eine Zahl aus IN. Demnach gibt es höchstens so viele Programme wie Zahlen in IN. Anders gesagt: Es gibt höchstens so viele berechenbare Probleme wie Zahlen in IN.
Andereseits: Wie viele Probleme, d.h. wie viele Funktionen f: IN →IN gibt es? Da $D_f \subseteq$IN ist, gibt es mindestens so viele Funktionen wie Teilmengen von IN. (Es gibt noch viel „mehr", da die Zuordnung der y-Werte zu einem Definitionsbereich ganz verschieden vorgenommen werden kann.)

Frage also: Wie viele Teilmengen hat IN? Betrachten wir, um einer Antwort näher zu kommen, zunächst endliche Mengen!

Aufgabe (eventl. Hausaufgabe):
Gegeben sei die Menge A ={a,b,c}
a) Bestimme die Menge aller Teilmengen von A! (Auch die leere Menge ist eine Teilmenge von A.)
b) Füge der Menge A ein weiteres Element hinzu und bestimme für die so entstehende Menge A' wiederum die Menge aller Teilmengen!
c) Eine Menge M habe n Elemente. Wie viele Teilmengen hat M? Begründung!

Die Aufgabe gibt den Schülern Gelegenheit zu einer „erholsameren" Form von Eigenaktivität als der in dieser Doppelstunde zuvor geleisteten und bereitet den Weg zum Begriff der Abzählbarkeit. Im Grunde könnte man ohne Umweg über endliche Mengen sogleich unendliche Mengen betrachten.

7./8. Stunde

Ziele: Begriff der Abzählbarkeit
Beweis, daß die Potenzmenge der natürlichen Zahlen überabzählbar ist

Besprechung der Hausaufgabe:
Die Menge A = {a,b,c} hat 8 Teilmengen. Die Menge A' = {a,b,c,d} hat außer den 8 Teilmengen, die A hat, weitere 8 Teilmengen, nämlich diejejenigen, die durch Hinzufügen des neu hinzugekommenen Elementes entstehen.

Allgemein: Durch Hinzufügen eines weiteren Elementes verdoppelt sich jeweils die Menge aller Teilmengen. Eine einelementige Menge hat 2 Teilmengen. Die Anzahl aller Teilmengen einer n-elementigen Menge ist somit 2^n ,die Menge aller Teilmengen verdient den Namen Potenzmenge, Bezeichnung: P(M).

Nach Besprechung der Hausaufgabe wird die Frage nach der Anzahl der Teilmengen von IN wieder aufgegriffen. Vorsicht bei unendlichen Mengen! Eine Teilmenge von IN ist z.B. die Menge G der geraden Zahlen. G ist eine echte Teilmenge von IN, insofern enthält G „weniger“ Elemente als IN. Andererseits lassen sich die Elemente G = {2,4,6,..} mit den natürlichen Zahlen 1,2,3,... durchnumerieren, sozusagen abzählen.

Wir sagen:
Eine Menge M ist „gleichmächtig zu IN“, wenn man die Elemente von M durchnumerieren kann: $M = \{m_1, m_2, m_3, ...\}$. Statt ‘M ist gleichmächtig zu IN’ sagen wir auch: ‘M ist abzählbar unendlich’.

Konkretisieren wir die oben gestellte Frage: Ist die Menge aller Teilmengen von IN abzählbar unendlich? Angenommen, sie wäre es. Dann könnte man die Teilmengen von IN durchnumerieren $P(IN) = \{T_1, T_2, T_3, ...\}$. Die Nummer i einer Teilmenge kann nun als Zahl in der Teilmenge T_i vorkommen oder nicht. Fassen wir alle Zahlen zusammen, die in ihrer entsprechenden Teilmenge nicht enthalten sind: $D = \{i \in IN: i \notin T_i\}$. D ist eine Zahlenmenge, also $D \subseteq IN$. Damit muß D eine der Teilmengen T sein, z.B. sei $D = T_n$. Was ist nun mit der Zahl n? Ist $n \in D$, so ist $n \notin T_n$. Folgt: $D \neq T_n$. Ist $n \notin D$, so ist $n \in T_n$. Folgt: $D \neq T_n$ Beides steht im Widerspruch zu $D = T_n$. Die Annahme ist demnach falsch.
Resultat: Die Potenzmenge der natürlichen Zahlen ist nicht abzählbar.
Zusammenfassung: Die Menge aller Programme ist eine unendliche Teilmenge von IN, also abzählbar. Die Menge aller Probleme ist mindestens so groß wie die Potenzmenge von IN, also nicht abzählbar. **Es gibt mehr Probleme als Programme.**

9./10. Stunde

Ziel: Beweis, daß das Halteproblem nicht berechenbar ist.

Nach dem Beweis, daß es unberechenbare Probleme gibt, stellt sich die Frage eines konkreten Beispiels. Ein solches Beispiel wird nun wie folgt in mehreren Schritten „entwickelt“: Sei P ein beliebiges Programm, das mit einer beliebig gewählten, festen Menge D von Eingabedaten belegt sei.

Problem: Entscheide, ob P eine Endlosschleife enthält oder nicht.
Das Problem ist eine Funktion $f: IN \rightarrow IN$, denn es ordnet einem beliebigen Programm $P \in IN$ eine der Zahlen aus {0,1} wie folgt zu:

$$f(P) = \begin{cases} 1, & \text{falls P eine Endlosschleife enthält} \\ 0, & \text{falls P keine Endlosschleife enthält} \end{cases}$$

Läßt sich dieses Problem berechnen?
Anders formuliert: Gibt es ein Programm H (Haltetester), das imstande ist, für ein beliebig, fest gewähltes Eingabeprogramm P den Funktionswert f(P) zu berechnen? Angenommen, es gäbe ein solches Programm H.
[und stellen wir uns vor, in H werde der berechnete Wert $f(P) \in \{0,1\}$ der Booleschen Variablen ‘Endlos’ zugewiesen. Dann könnten wir die Ausgabe dieses Ergebnisses wie folgt erzeugen:

```
IF Endlos THEN
  writeln('Das geprüfte Programm enthält eine Endlosschleife.')
```

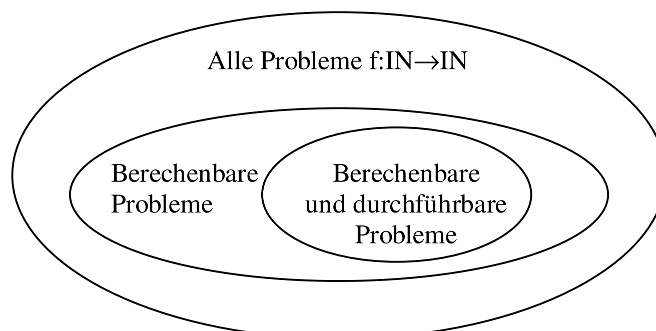
```
ELSE
  writeln('Das geprüfte Programm enthält keine Endlosschleife.');]
```

Dann könnte man H, falls es diese Eigenschaft nicht bereits hat, folgende Eigenschaft hinzufügen: H wird in eine Endlosschleife geschickt, falls das Programm P, auf das es angewendet wird, terminiert. H wird beendet, falls das Programm P, auf das es angewendet wird, eine Endlosschleife enthält.

```
[z.B. so:
IF Endlos THEN
  writeln('Das geprüfte Programm enthält eine Endlosschleife.')
ELSE
  BEGIN
    writeln('Das geprüfte Programm enthält keine Endlosschleife.');
    WHILE TRUE DO; {Endlosschleife}
  END;]
```

H ist ein Programm, wenden wir daher H auf (P=) H an:
Hat (die Eingabe P=) H eine Endlosschleife, so terminiert (der Haltetester) H.
Hat (die Eingabe P=) H keine Endlosschleife, so wird (der Haltetester) H in eine Endlosschleife geschickt.
Wir erhalten jeweils einen Widerspruch. Die Annahme, es gäbe einen Haltetester, ist falsch.

Zur Sicherheit zwei Hinweise:
Das Problem heißt Halte-Problem. Könnte es außer Endlosschleifen auch andere Ursachen eines Nicht-Anhaltens geben? Antwort: Nein, denn ein Programm ist eine endliche Folge von Anweisungsblöcken, die in endlicher Zeit abgearbeitet sein muß, falls nicht mindestens einer dieser Anweisungsböcke eine unendliche Folge von Einzelanweisungen darstellt.
Die Pascal-Fragmente in obigem Beweis bewerkstelligen lediglich die Ausgabe. Der Knackpunkt ist natürlich die Belegung der Variablen Endlos. Der Beweis zeigt: Es gibt keinen Algorithmus zur Belegung dieser Variablen.
Dem Beweis eines nicht berechenbaren Problems folgt eine Zusammenfassung in Form eines Mengendiagramms:



Das Thema 'Nicht-berechenbare Probleme' ließe sich vertiefen.
Gibt es weitere nicht-berechenbare Probleme?

Die Nicht-Berechenbarkeit der beiden folgenden Probleme wird bewiesen, indem man zeigt, das im gegenteiligen Fall das Halteproblem berechenbar wäre.

Totalitäts-Problem

Machen wir uns bewußt, das Halteproblem geht von einem beliebig, fest gewähltem Programm aus, das mit einem beliebig, fest gewählten Menge von Eingabedaten versehen ist.
Davon zu unterscheiden ist das folgende sog. Totalitäts-Problem:

Sei P ein beliebiges Programm
Problem: Stoppt P <u>für alle</u> Eingaben D oder nicht?
Man sagt: P ist total, falls P(D) für alle Eingaben D stoppt.

Angenommen, das Totalitäts-Problem wäre berechenbar. Dann ließe sich zu beliebig, festen P und D berechnen, ob P(D) stoppt oder nicht und daraufhin ein spezielles Programm bilden, das P(D) simuliert (d.h. zur gleichen Ausgabe führt, falls P(D) stoppt oder nicht stoppt, falls P(D) nicht stoppt), hierbei aber seine Eingabe I, wie immer sie sein mag, ignoriert. Nennen wir dieses spezielle Programm *Spaßig-PD(I)*.

Wenn P(D) stoppt, so stoppt auch Spaßig-PD, und zwar laut Definition <u>für alle</u> Eingaben I, d.h. Spaßig-PD ist total. Mit Hilfe von Spaßig-PD ließe sich nun wie folgt ein Algorithmus für das Halteproblem formulieren.
Haltetester(P,D):
Erstelle den Text für das Programm Spaßig-PD {setzt voraus, daß es einen Algorithmus für das Totalitäts-Problem gibt}
Falls Spaßig-PD total ist
    dann gib „P(D) stoppt“ aus und stoppe
    sonst gib „P(d) stoppt nicht“ aus und stoppe

„Ein anderes nicht-berechenbares Problem ergibt sich bei Unternehmen, die einen neuen Computer installieren. Stellen wir uns vor, ein Unternehmen besitzt eine Vielzahl von Programmen, die jahrelang fehlerfrei auf den alten Computern gelaufen sind. Die Programmierer haben gerade einen Satz neuer Programme für die gleichen Aufgaben auf dem neuen Computer geschrieben. Wie kann das Unternehmen sicher sein, daß die neuen Programme ebenso zuverlässig laufen wie die alten?

Klar, das Unternehmen benötigt einen Algorithmus, der jeweils zwei Programme nimmt, ein altes und ein neues, und sie vergleicht, um festzustellen, ob sie die gleiche Aufgabe erfüllen (d.h. für die gleiche Eingabe erzeugen sie die gleiche Ausgabe). Dies ist das *Äquivalenz*-Problem. Bedauerlicherweise ist das Äquivalenz-Problem nicht-berechenbar.“ ([Go/Li86, S.87]

Dem folgenden Beweis liegt folgende Idee zugrunde:
Wenn es einen Algorithmus gäbe, der für zwei beliebige Programme feststellen könnte, ob sie „die gleiche Aufgabe“ (s.o.) erfüllen, so könnte man mit ihm einen Algorithmus angeben, der das Totalitäts-Problem berechnet.

Beweis:
Sei P ein beliebiges Programm, D eine beliebige Eingabemenge. Fügen wir dem Programm P(D) eine weitere Anweisung zu, wie etwa „Gib 13 aus“, und nennen wir dieses spezielle Programm *Spaßig-P(D)*.

Spaßig-PD:
  P(D)
  „Gib 13 aus“.

Spaßig-PD simuliert P(D) und gibt zusätzlich die Zahl 13 aus, falls P(D) stoppt. Betrachten wir daneben ein einfältiges Programm, das stets „13“ ausgibt.

Einfältig(D)
{Die Eingabe D wird ignoriert}
  Gib „13“ aus.

Zu fragen, ob *Spaßig-P* äquivalent zu *Einfältig* sei, bedeutet das gleiche, wie zu fragen, ob P total sei. Denn stoppt P bei jeder Eingabe, so gibt Spaßig-P bei jeder Eingabe „13“ aus. Gibt es aber irgendeine Eingabe, bei der P nicht stoppt, dann stoppt Spaßig-P auch nicht und gibt bei dieser Eingabe auch nicht „13“ aus. Gäbe es einen Algorithmus für das Äquivalenz -Problem, so ließe sich wie folgt ein Algorithmus für das Totalitäts-Problem formulieren:

Erstelle den Text für das Programm Spaßig-P
Erstelle den Text für das Programm Einfältig
Falls Spaßig-P und Einfältig äquivalent sind
    dann gib aus „P ist total“
    sonst gib aus „Pist nicht total“
Die bislang behandelten, nicht-berechenbaren Probleme beziehen sich auf Fragen zu Algorithmen. Golschlager und Lister nennen auch solche anderer Natur, u.a. das folgende:
„Für eine beliebige Gleichung (wie $(a+1)^2 + (a+1)^3 = (a+1)^4$) möchte man bestimmen, ob die Gleichung durch irgendeine Ganzzahl gelöst werden kann. Dieses Problem heißt Problem der *Diophantischen Gleichung*. Es ist bekannt, daß kein Algorithmus zur Lösung beliebiger Diophantischer Gleichungen existiert.“ [Go/Li86, S.89]

2. Unterrichtseinheit: Automaten

Didaktische und methodische Überlegungen

Automaten begegnen wir im täglichen Leben auf Schritt und Tritt. Waschmaschine, Getränke- und Warenautomaten, Fahrkartenautomaten sind Beispiele für endliche Automaten. Neben diesen Beispielen aus dem Alltag spielt die Modellierung von Automaten zur Mustererkennung (Texterkennung, Textverarbeitung, Compiler) eine große Rolle in der Informatik.

Unter Automaten versteht man Maschinen, die - erst einmal in Gang gesetzt - ohne weiteren Eingriff des Menschen arbeiten. Automaten dienen zur Verarbeitung von Information. Auf eine bestimmte Eingabe, liefert der Automat eine Ausgabe. Ein- und Ausgabe bestehen aus Signalen (Zeichen und Daten). Jeder Automat muß, um seine Funktion aufrechterhalten zu können, einen Informationsspeicher besitzen. Das Verhalten der Maschine wird nach einer Vorschrift (von einem Programm) gesteuert. Da der Automat für eine bestimmte Aufgabe entwickelt wurde, ist das Programm nicht von der Maschine abtrennbar: Maschine und Programm sind eins [Bau 90, Bau 93]. Technisch werden einfache Automaten mit digitalen Schaltungen realisiert, z.B. Speicher, Halbaddierer.

Informatik als Wissenschaft behandelt Verarbeitung von Daten durch Automaten. Der Automat als mathematisches Modell dient hierbei zur Beschreibung und Präzisierung dieses Vor-

gangs [KSP 94]. Automaten, Sprachen und Algorithmen sind zentrale Begriffe in der theoretischen Informatik.

Der Automat wird durch ein technisches Gerät verwirklicht, das Eingabedaten unter Beachtung seines inneren Zustands zu Ausgabedaten transformiert. In der theoretischen Informatik unterscheidet man zwischen deterministischen Automaten (DEA) und nichtdeterministischen Automaten (NEA). Ein deterministischer Automaten kann auf die Eingabe eines Datums nur in genau einen Zustand übergehen, während eine nichtdeterministischer Automat durch Eingabe eines Datums unvorhersehbar in verschiedene Zustände übergehen kann. Bei einem nichtdeterministischen Automaten kann es auf die Eingabe eines Datums mehrere Übergangsmöglichkeiten in verschiedene Zustände geben.

In der Realität gibt es nur deterministische Automaten, weil auf eine bestimmte Eingabe immer eine feststehende Ausgabe folgt. Selbst Glücksspielautomaten, die den Anschein von Nichtdeterminismus erwecken, werden durch Zufallszahlen zu einer festgelegten Ausgabe geführt. Der Vorteil nichtdeterministischer Automaten liegt in der einfachen Darstellungsweise und ihrer Beschreibung in Form von Graphen. Ein zentraler Satz der theoretischen Informatik besagt, daß jeder NEA in einen DEA übergeführt werden kann.

Mit den endlichen Automaten werden in dieser Unterrichtseinheit die einfachsten Modelle informationsverarbeitender Maschinen kennengelernt. Hierbei werden die prinzipiellen Grenzen der Automaten aufgezeigt und begründet. Auch durch dies Einheit zieht sich der Begriff der Berechenbarkeit. Probleme, die mit endlichen Automaten modelliert werden können, sind berechenbar.

Planung der Unterrichtseinheit

Da Automaten aus unserm Alltag nicht wegzudenken sind, halte ich es für sinnvoll, daß Thema mit einem Einfachen Beispiel aus der Erfahrungswelt der Schüler zu motivieren. Hier bieten sich Cola-Automat oder Fahrkartenautomat an. Die den Schülern bekannten Fahrkartenautomaten verfügen über recht viele Funktionen, die man auf eine Minimum reduzieren müßte um dieses als Einführungsbeispiel zu nehmen (etwa wie bei [Bau 93] S.173). Der Getränkeautomat ist eine einfacheres Beispiel, das sich wegen der wenigen Eingabemöglichkeiten gut eignet. Hier können die Begriffe *Eingabealphabet, Ausgabealphabet, Zustandsmenge, Übergangsfunktion* und *Ausgabefunktion* erarbeitet werden. Eine graphische Veranschaulichung des Automaten mittels gerichteter Graphen kann bereits in dieser Anfangsphase erfolgen. Außerdem kann eine Zustandstabelle zur vollständigen übersichtlichen Beschreibung eines Automaten aufgestellt werden. Ein weiterer Automat sollte zur Vertiefung besprochen werden. Um kein zu gleichartiges Beispiel zu bearbeiten entschieden wir uns für den „Hund als Automat“ [Jae 86].

Eine weitere wichtige Anwendung von Automaten ist die Text- und Mustererkennung, wie sie z.B. in Textverarbeitung und Compilerbau zum Tragen kommt. Sollte genügend Zei im Unterricht zur Verügung stehen, kann eine einfacher Compiler (LOGO für Arme, siehe ausführliche Beschriebung im Kapitel „Sprachen und Grammatiken“) programmiert werden. Sonst bietet sich dieses Thema auch für eine Hausarbeit an.

Im Rahmen der Mustererkennung findet man erste einfache Beispiele für Probleme, die mit NEAs weitaus unkomplizierter dargestellt werden können als mit DEAs. Zur praktischen Rea-

lisierung muß man diese dann in einen DEA überführen. Als zentraler Satz in der Automatentheorie muß hier erwähnt werden, daß man zu jedem NEA einen DEA konstruieren kann.

Zunächst könnte ein Texterkennungsproblem gestellt werden, welches die Schüler deterministisch lösen können. Damit die Einführung von nichtdeterministischen Automaten den Schülern nicht zu sehr „übergestülpt“ wird, muß dann Problem zur Texterkennung gestellt werden, bei der ein NEA einfach hingeschrieben werden kann, dessen deterministische Lösung auch nach längerem Probieren kaum gefunden wird.

Zur Überprüfung der entworfenen Automaten kann eine Simulationsumgebung genutzt werden. Hier wird die Übergangsfunktion des Automaten eingegeben, so daß jedem Paar (Zustand, Eingabe) eine Folgezustand zugeordnet wird. Über ein Menü kann dann in das Eingabeband die Eingabenfolge festgelegt werden. Das Simulationsprogramm arbeitet die Eingabezeichen nacheinander ab und zeigt die durchlaufenen Zustände in einem Ausgabeband an. Die Simulationsumgebung ist erhältlich auf dem ftp-server der Universität Frankfurt, ftp://ftp.rz.uni-frankfurt.de/pub/lwb/, unter dem Dateinamen automat.zip.

Das Thema kann zusätzlich abgerundet werden mit gesellschaftlichen Aspekten und Fragestellungen. Das Verhältnis zwischen Mensch und Maschine kann beleuchtet werden, sowie die Frage, was Computer prinzipiell nur leisten können. Eventuell könnte sich eine Diskussion zu kypernetischen Fragen, neuronalen Netzen, selbstlernenden System anschließen und zur nächsten Unterrichtsreihe überleiten.

In dieser Unterrichtseinheit ist nicht geplant, auf die technische Realisierung von Automaten mit digitalen Schaltungen einzugehen. Auch wird kein geschichtlicher Abriß zur Automatenentwicklung und Entstehung von Computern gegeben. Beides könnte anstelle einer zweiten Klausur als Hausarbeit von Schülern bearbeitet werden. Innerhalb dieser Unterrichtseinheit würde insbesondere das Besprechen von logischen Schaltungen den Rahmen sowohl in zeitlicher wie in inhaltlicher Sicht sprengen.

Lernziele

Die Schüler sollen

- erkennen, was mit einem endlichen Automaten berechenbar ist
- endliche Systeme durch Automaten beschreiben können
- Unterschiede zwischen deterministischen und nichtdeterministischen Automaten kennenlernen
- NEA in DEA umwandeln können
- Anzahl der Zustände eines DEA angeben können
- wissen, daß bei der Umwandlung eines NEA in einen DEA die Potenzmenge die Maximalzahl der möglichen Zustände des DEA angibt
- Automaten graphisch und in Form von Tabellen darstellen können
- Simulationsumgebung für Automaten benutzen können
- Anwendung für Texterkennung und Interpreter kennenlernen
- wissen, daß reguläre Sprachen die Sprachen endlicher Automaten sind
- die Grenzen endlicher Automaten kennen

Möglicher Verlauf der Unterrichtseinheit

1. Doppelstunde: Einführung deterministische Automaten

Einführung in die Automatentheorie anhand von einfachen endlichen Automaten. Beispiele hierfür sind Fahrkartenautomat und Getränkeautomat. Wir schlagen vor, den Getränkeautomaten (Cola-Automat) als einführendes Beispiel zu wählen:

Der Getränkeautomat kann Cola und Limonade ausgeben. Beide Getränke kosten 1,50DM, es können Münzen zu 1DM und zu 0,50 DM eingeworfen werden. Wird der Betrag von 1,50DM überschritten, so fällt die Münze ins Geldfach, bei korrektem Geldeinwurf kann zwischen den beiden Getränken oder der Geldrückgabe gewählt werden. Sonst führt Drücken der Korrekturtaste zu Geldrückgabe [Jae 86]. Möglicherweise kann zur Vereinfachung des Automaten auf die Limonadentaste verzichtet werden.

Die Arbeitsweise des Automaten kann zunächst mit den Schülern erarbeitet werden. Nachdem eine verbale Beschreibung des Automaten formuliert wurde ist es möglich, den Automaten genauer zu untersuchen. Die Eingaben, Ausgaben und Zustände werden herausgearbeitet:

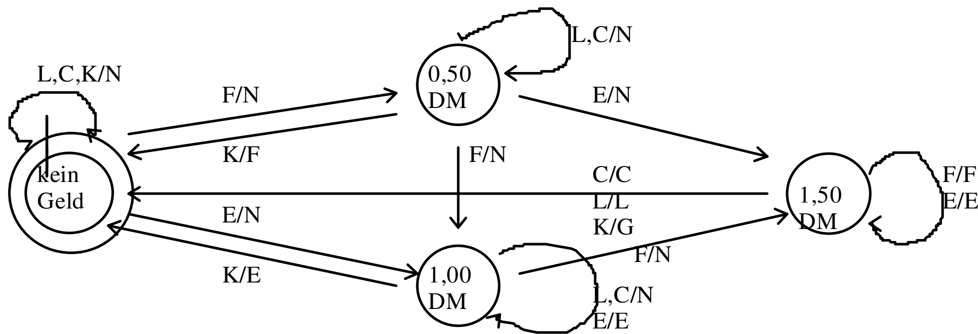
Eingaben sind: 0,50DM, 1DM, 2 Wahltasten Cola/Limonade, Korrekturtase
Zustände sind: kein Geld, 0,50DM, 1DM, 1,50DM
Ausgaben sind: 0,50DM, 1DM, 1DM und 0,50DM, Cola, Limonade, Nichts
Die Ausgabe 1DM kann hier in zwei physikalischen Realisierungen auftreten: ein Geldstück à 1DM oder zwei Geldstücke à 0,50DM. Der Anfangszustand ist gleich dem Zielzustand „kein Geld“.

Je nach dem Kenntnisstand der Schüler kann nun zunächst eine Zustandstabelle oder das Zustandsdiagramm erarbeitet werden. Am Ende der Doppelstunde sollten beide Darstellungsmöglichkeiten den Schülern bekannt sein.

Die Zustandstabelle:

Eingabe Zustände	0,50 DM	1,00 DM	Taste Cola	Taste Limo	Korrektur
kein Geld	0,50 DM / N	1,00 DM / N	kein Geld / N	kein Geld / N	kein Geld / N
0,50 DM	1,00 DM / N	1,50 DM / N	0,50 DM / N	0,50 DM / N	kein Geld / 0,50 DM
1,00 DM	1,50 DM / N	1,00 DM / 1 DM	1,00 DM / N	1,00 DM / N	kein Geld / 1,00 DM
1,50 DM	1,50 DM / 0,50DM	1,50 DM / 1 DM	kein Geld / Cola	kein Geld / Limo	kein Geld / 1,50 DM

Das Zustanddiagramm hat folgende Gestalt:



Die Abkürzungen bedeuten: E(Einwurf 1 DM), F(Einwurf 0,50 DM), K(Korrekturtaste), G(Geld zurück), L(Limonade), C(Cola), N(keine Ausgabe)

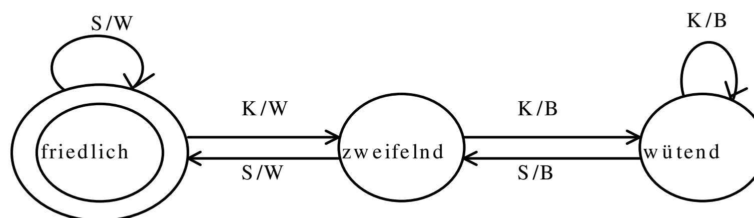
An dem Beispiel des Getränkeautomaten können die Fachworte Eingabezeichen (Eingabealphabet), Ausgabezeichen (Ausgabealphabet) und innere Zustände (Zustandsmenge) gut erarbeitet und definiert werden.

Als weiteres, nicht ganz so ernst zu nehmendes Beispiel für einen Automaten haben wir den „Hund als Automat" ausgewählt [Jae 86]. Der Automat Hund wird wie folgt charakterisiert:

Eingabe: streichenl, knuffen
Zustand: friedfertig, zweifelnd, wütend
Ausgabe: wedeln, bellen

Ist der Hund friedfertig, so folgt auf die Eingabe „streicheln" die Ausgabe „wedeln". Wird der friedfertige Hund geknufft, wedelt er weiter, geht aber in den Zustand zweifelnd. Wird der zweifelnde Hund gestreichelt, so wedelt er und wird friedfertig, wird er nochmals geknufft, so bellt er und wird wütend. Der wütende Hund readiert auf knuffen mit bellen und bleibt wütend, auf streicheln bellt er weiter, wird aber friedfertig.

Das Zustandsdiagramm hat folgende Gestalt:



Hierbei bedeuten die Abkürzungen S(steicheln), K(knuffen), W(wedeln), B(bellen). Weitere Beispiele finden sich auf den Übungsblättern.

2. Doppelstunde: Einführung nichtdeterministische Automaten

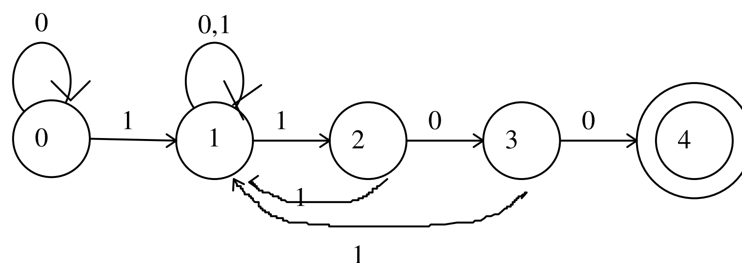
Zur Wiederholung der Eigenschaften von Automaten und zum Üben der gaphischen wie tabellarischen Darstellungsweise wird ein weiteres Anwendungsbeispiel für DEAs behandelt. Da

Automaten in der Mustererkennung (Textverarbeit und Compiler) eine große Rolle spielen, sollte ein entsprechendes Beispiel gewählt werden. Aufgaben hierzu finden sich auf den Übungsblättern.

Je nach zur Verfügung stehendem Zeitumfang für die Unterrichtseinheit, kann an dieser stelle eine Miniprogrammiersprache für graphische Programmierung als Anwendung von DEAs im Compilerbau erstellt werden. Es eignet sich besonders ein MiniLogo (Logo für Arme), das Turtel-Graphikbefehle ausführt [Mod 92]. Steht im Rahmen der Unterrichtseinheit nicht genügend Zeit für diese Programmieraufgabe zur Verfügung, kann sie als Hausarbeit an Stelle einer zweiten Klausur vergeben werden. Auch eignet sich diese Aufgabe im Rahmen von Projektarbeit (genaue Beschreibung des Mini-LOGO erfolgt im Kapitel „Sprachen und Grammatiken).

Aufgaben zur Mustererkennung eigen sich gut, um nichtdeterministische Automaten einzuführen. Hierzu wählen wir folgendes Beispiel: Es soll erkannt werden, ob eine Folge der Zahlen Null und Eins mit einer Eins beginnt und auf 100 endet. Die Formale Schreibweise für dieses Muster lautet 1 {0,1}* 100. Zwischen der ersten Ziffer 1 und dem Ende 100 können die Ziffern 0 und 1 in beliebiger Reihenfolge beliebig oft au ftreten.

Zunächst sollen die Schüler probieren, den deterministischen Automaten zu konstruieren. Eine Lösung ist nur schwer zu finden, den Schülern wird wohl nach 20 Minuten Porbierzeit keine Lösungsmöglichkeit einfallen. Nichtdeterministische läßt sich der Automat in einem einfachen Graphen darstellen:

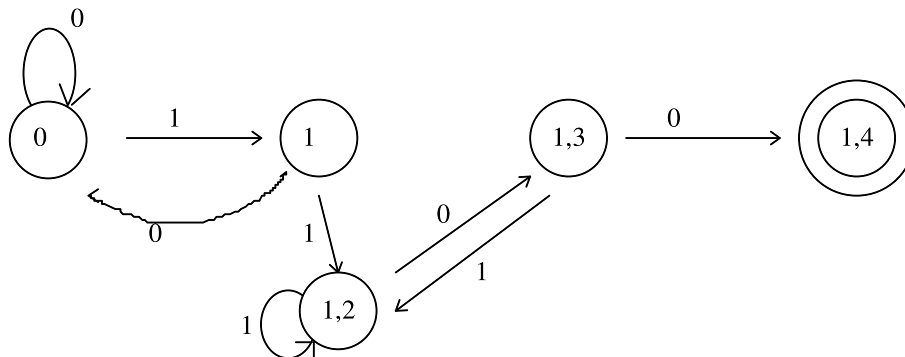


3. und 4. Doppelstunde: Umwandlung von Nea in Dea

An mehreren Beispielen wird gezeigt, wie über die Zusammenfassung von möglichen Zuständen in Teilmengen ein Nea in einen Dea umgewandelt werden kann. Hierbei sollte die Darstellung sowohl tabellarisch als auch graphisch erfolgen. Die Zustandstabelle des DEA für das Mustererkennungsbeispiel aus der letzten Stunde sieht wie folgt aus:

Eingabe Zustände	0	1
0	0	1
1	1	(1,2)
1,2	(1,3)	(1,2)
1,3	(1,4)*	(1,2)

Hierbei sind die mit * gekennzeichneten Zustände Endzustände. Der Graph hat folgendes Aussehen:



An mehreren Beispielen wird das Umformen von NEAs in DEAs geübt. Zunächst wird deutlich, daß beim Überführen des NEAs in einen DEA mehr Zustände eingeführt werden müssen. Aus dem Umwandlungsverfahren erkennt man, daß dieses für jeden möglichen NEA anwendbar ist. Ohne einen Beweis vorzuführen, wird daher den Schülern mitgeteilt, daß jeder NEA in einen DEA umgewandelt werden kann: „zu jedem NEA gibt's 'nen DEA".

Mit den Grundkenntnissen aus der Stochastik kann erklärt werden, daß die Maximalzahl der Zustände gerade der Potenzmenge entspricht. Für drei Zustände A={a,b,c} ist die Potenzmenge P(A)={ {a,b,c}, {a,b}, {a,c}, {b,c}, {a}, {b}, {c}}. Sie besteht also aus 8 Zuständen |P(A)|=8. Fügt man eine weiteres Element d zu A hinzu, so verdoppelt sich die Anzahl der Teilmengen. Allgemein gilt: $|P(A)|=2^{|A|} > |A|$. Somit wächst die Anzahl der Zustände exponentiell bei der Umwandlung eines NEA in einen DEA. Allerdings kann ein Teil der Zustände als überflüssig eliminiert werden. An dieser Stelle kehren wir wieder zu dem Begriff der Berechenbarkeit zurück.

5. Doppelstunde: Die Grenzen endlicher Automaten

Die Einheit wird abgerundet durch Überlegungen zu den Grenzen endlicher Automaten. Gleichzeittig wird die folgende Einheit zum Theam „Sprachen und Grammatiken“ eingeleitet.

Es soll gezeigt werden, daß von einem endlichen Automaten nur Sprachen erkannt werden, die keine beliebig tief verschachtelten Strukturen enthalten. Solche Verschachtelungen findet man in jeder Programmiersprache als „Klammerpaare“ wie BEGIN...END, REPEAT...UNTIL.

Baumann ([Bau 93], Seite 225) schlägt vor, die Schüler einen Automaten entwickeln zu lassen, der die Sprache $\{a^n b^n \mid 1 <= n <= N\}$ erkennt für a) N=3, b) N=7.

Zu jedem N kann man einen Automaten konstruieren, allerdings findet sich kein Automat, bei dem die Anzahl der Schachtelungen nicht beschränkt ist. Ein Automat, der nur endlich viele Zustände hat, ist also den Anfordrungen an einen Compiler nicht gewachsen. Hierzu benötigt man Automaten mit „Gedächtnis“, Kellerautomaten.

3. Unterrichtseinheit: Grammatiken und Sprachen

Didaktische und methodische Anmerkungen

In dieser Unterrichtseinheit können die Schüler das Modell des Automaten zur Anwendung bringen, indem sie untersuchen, welchen Regeln Programmiersprachen folgen und wie man diese auch formal auf Korrektheit überprüfen kann. Dies kann auf verschiedene Weise erfolgen. Man kann sich auf die Darstellung von Scannern und Parsern durch geeignete Automaten beschränken oder aber die bei formalen Sprachen und Grammatiken übliche Darstellung durch Terminal- und Nichtterminalsymbole und das entsprechende Regelsystem zur Erzeugung der Sprache verwenden. Falls genügend Zeit zur Verfügung stehen, sollte der letzte Weg zumindest an einem einfachen Beispiel gegangen werden. Der Schüler soll ja auch hier mit den in der Fachwissenschaft üblichen „Beschreibungs- und Darstellungsformen“ ([KSP 94] , S.12) vertraut gemacht werden.

Dadurch wird ein Bezug hergestellt zu dem Unterricht der vergangenen Halbjahre und die Schüler können besser verstehen, daß Programmiersprachen bestimmten Einschränkungen unterworfen sind, die z.T. für natürliche Sprachen nicht gelten. Dies kann dann dazu führen, daß Schüler selbst eine einfache „Programmiersprache“ entwerfen, was als Vorbereitung zu Überlegungen zum Compilerbau dienen könnte.

Andererseits erährt das Modell des Automaten eine Erweiterung, daß die Konzepte des Kellerautomaten und der Turingmaschine eingeführt werden. Über diese Konzepte kann wieder an die erste Unterrichtseinheit angeknüpft werden, indem auch hier wieder über Berechenbarkeit und Entscheidbarkeit reflektiert werden kann und die Turingmaschine als Modell für berechenbare Probleme erkannt werden soll.

Lernziele

Die Schüler sollen

- Grundlagen von Scannern und Parsern kennenlernen

- den Kellerautomaten kennenlernen als Möglichkeit, durch Einführen eines Gedächtnisses, die Grenzen des endlichen Automaten zu erweitern
- die Bedeutung kontextfreier Grammatiken für die Entwicklung von Programmiersprachen kennen
- die Turingmaschine als Erweiterung des Automaten kennen
- wissen, daß alles was berechenbar ist, mit der Turingmaschine machbar ist
- den Algorithmusbegriff kennen

Reguläre Ausdrücke und endliche Automaten

Reguläre Ausdrücke bieten die Möglichkeit, die von Automaten akzeptierte Sprache einfach und übersichtlich aufzuschreiben. *Sprachen, die von einem endlichen Automaten akzeptiert werden, heißen regulär.* Sie lassen sich durch solche regulären Ausdrücke beschreiben. Es gibt eine Äquivalenz zwischen den von endlichen Automaten akzeptierten Sprachen und den mittels regulärer Ausdrücke erzeugten Mengen. Deswegen werden die *Sprachen endlicher Automaten* auch *reguläre Mengen* genannt.

Der Begriff *regulärer Ausdruck* wird induktiv wie folgt definiert:
A sei ein Alphabet, dann ist

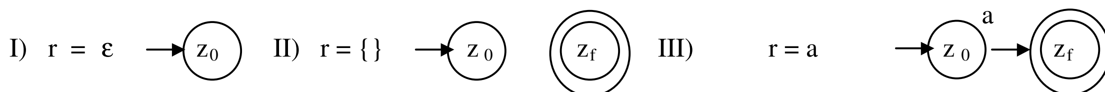
- jedes $a \in A$ ein regulärer Ausdruck,
- wenn s und r reguläre Ausdrücke sind, dann sind es auch
- - {s,r} (Vereinigung, „s oder r“)
- - rs (Aneinanderreihung)
- - r* (Sternbildung, r beliebig oft wiederholt, also auch 0-mal wiederholt)

Der *-Operator hat hierbei höhere Priorität als die Reihung und Vereinigung, Reihung hat höhere Priorität als Vereinigung. Überflüssige Klammern werden häufig weggelassen.

Beispiele: Sei A={a,b}

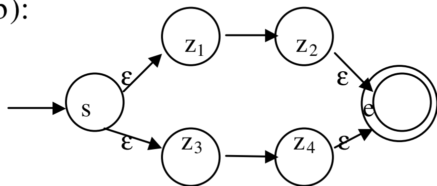
- ab* bezeichnet die Menge aller Worte, die mit a beginnen und mit einer beliebigen Anzahl von b's enden
- a*bba* ist die Menge aller Worte, die genau zwei b's enthalten.
- (a,b)* enthält alle Zeichenketten, die aus a oder b gebildet werden können.
- (a,ab)* ist die Menge aller Worte aus a's und b's, die mit einem a beginnen und keine zwei aufeinanderfolgenden b's enthalten, während (b,ε)(a,ab)* auch mit einem b beginnen darf. (ε ist das leere Wort)

Um nun zu jedem regulären Ausdruck einen EA konstruieren zu können, stellt man sich Automatenbausteine für die Grundelemente der regulären Ausdrücke zusammen und kombiniert diese den Anforderungen zu Automaten.



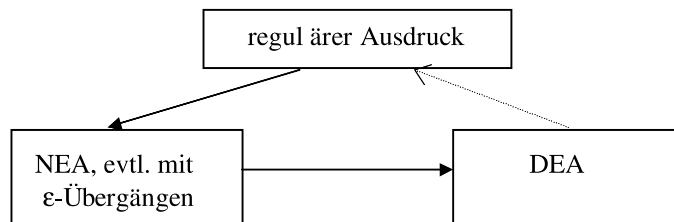
Formal gesehen verfährt man nun wie folgt: Durch Hinzufügen von neuen Zuständen und ε-Übergängen werden diese Bausteine zu neuen Automaten zusammengesetzt.

Bsp.: r = (a,b):



ab stellt sich als zwei Automaten vom Typ III, die durch einen ε-Übergang miteinander verbunden sind, ergänzt um Start- und Zielzustand mit ε-Übergängen.

Um die Äquivalenz zwischen regulären Ausdrücken und Automaten zu zeigen, muß noch ein Verfahren angegeben werden, das zeigt, wie zu einem DEA ein regulärer Ausdruck bestimmt wird, der die akzeptierte Sprache beschreibt. Der Äquivalenzbeweis folgt dann dem Schema:



Der waagrechte Pfeil ergibt sich unmittelbar, der linke Pfeil folgt aus dem obigen Konstruktionsschema, gezeigt werden muß noch die Existenz des gestrichelt gezeichneten Pfeiles.

Die Konstruktionsidee, die hier nicht ausgeführt werden soll, beruht darauf, daß der reguläre Ausdruck alle möglichen Wege beschreiben muß, die von dem Startzustand s zu dem Zielzustand z_e führen. Für jeden der möglichen Teilwege läßt sich ein regulärer Ausdruck angeben. Um systematisch vorzugehen, bildet man die regulären Ausdrücke für alle Wege, die von einem Knoten zu einem anderen führen und dabei durch bestimmte Knoten hindurchgehen. Hat man alle diese regulären Ausdrücke gefunden, ergibt sich daraus auch der gesuchte. (Einzelheiten s. z. B. [Hop90, S. 34ff])

Sprache / Grammatik

In diesem Abschnitt geht es in erster Linie nicht um den Automaten, der eine Eingabefolge akzeptiert, sondern um die Struktur der Eingabe, die den Automaten in einen Endzustand versetzt. Um einen Automaten zu entwerfen, benötigt man ein gegebenes Regelwerk, das uns zeigt, die richtige Sätze der beschriebenen Sprache aufgebaut sind. Diese durch Regelwerke beschriebenen Sprachen lassen sich klassifizieren, um dann entsprechende Automaten zu konstruieren, die die Sprache akzeptieren.

Mit Hilfe von Sprachanalye und dem Aufbau von Grammatiken können die Schüler einen einfachen Zugang zum Übersetzungsaufbau finden. Sprachkonstrukte, die sich regulär beschreiben lassen, sind deshalb analysierbar, und man kann auch eine Maschine angeben, die Sätze dieser Art analysiert. Als Anwendung dieser Einheit steht die Entwicklung von „Logo für Arme“ (siehe Anhang). Die Schüler sollen hierbei die Aufgaben eines Compilers, d. h. von Scanner (lexikalische Analyse) und Parser (Syntaxanalye) kennenlernen.

1. Sprache
Zur Einführung von Sprache, soll die Struktur einer Sprache am Beispiel der deutschen Sprache verdeutlicht werden.

Definition:
A sei ein Alphabet, d.h. eine endliche Menge von Zeichen. A* ist die Menge der Zeichenreihen über A.

Beispiel:
A = {0,1}
A* = {ε, 0, 1, 00, 01, 10, 11, 000, ...}

Eine Sprache über einem Alphabet A ist eine Teilmenge von A* . Die Symbole einer Sprache werden die Terminalzeichen genannt Die deutsche Sprache besteht beispielsweise aus einer Aneinanderreihung von deutschen Worten. Die einzelnen Worte sind die Terminalzeichen der Sprache.

Um eine Sprache zu analysieren, sind die mehrere Schritte nötig. Diese werden im folgenden beschrieben und zugleich an einem Beispiel erläutert (Deutsche Sprache als Programmiersprache)

1.1 lexikalische Analyse
In der Menge aller möglichen Zeichenreihen interessiert man sich für diejenigen, die gewissen lexikalischen Bedingungen genügen (etwa die in einem Duden). Damit beschäftigt sich die lexikalische Analyse.

Deutsche Sprache:
„a234sadf&%“ ist lexikalisch falsch

1.2 syntaktische Analyse
Aus lexikalisch richtigen Zeichenreihen („Worten“) lassen sich Programme („Sätze“) aufbauen, deren Struktur gewisser grammatischen Regeln genügen. Um diese Struktur geht es bei der syntaktischen Analyse (oder kurz: Syntaxanalyse)

Deutsche Sprache:
„Hund und baut alle sind“ ist lexikalisch richtig aber syntaktisch falsch

1.3 semantische Analyse
Sehr häufig erlaubt eine Syntax eine Formulierung von Sätzen, die zwar syntaktisch richtig, aber aufgrund des Kontexts widersinnig oder bedeutungslos sind. Das Aussondern solcher Sätze wird als „semantische Analyse“ bezeichnet.

Deutsche Sprache:
„Der Hund fliegt kluges Wasser“ ist syntaktisch richtig, macht aber keinen Sinn

Diejenigen Sätze einer Sprache, die syntaktisch richtig sind und zudem den Kontextbedingungen genügen, wird eine Bedeutung beigelegt. Dieser Teil einer Sprachbeschreibung wird Semantik genannt.

Deutsche Sprache:
„Der Hund bellt“ ist semantisch korrekt

Alle möglichen Reihungen von Terminalsymbolen bilden sicherlich, wie man eben gesehen hat, noch keine Sprache. Um eine Sprache zu beschreiben, benötigt man also gewisse Regeln, diese

bilden zusammen mit den Terminalsymbolen eine Grammatik. Bezeichnet man eine Grammatik mit G, so nennt man die entsprechende Sprache L(G).
[Mod 1988, S. 123]

2. Grammatik
Eine Regelgrammatik kann formal folgendermaßen beschrieben werden:
T = endliche Menge von terminalen Symbolen
N = endliche Menge von nichtterminalen Symbolen
$S \in N$ = Startsymbol
R = einem endlichen System von Regeln, die es gestatten, eine Symbolfolge durch eine andere zu ersetzen

Beispiel 1: (Teil der deutschen Grammatik)
N = {Satz, Subjekt, Prädikat, Artikel, Substantiv, Verb}
T = {der, die, das, ein, einer, ball, heimat, kind, fliegt, schreit, winkt}
S = Satz
R = {Satz → Subjekt Prädikat,
Subjekt → Artikel Substantiv,
Prädikat → Verb,
Artikel → der | die | das | ein | einer
Substantiv → ball | heimat | Kind,
Verb → fliegt | schreit | winkt}

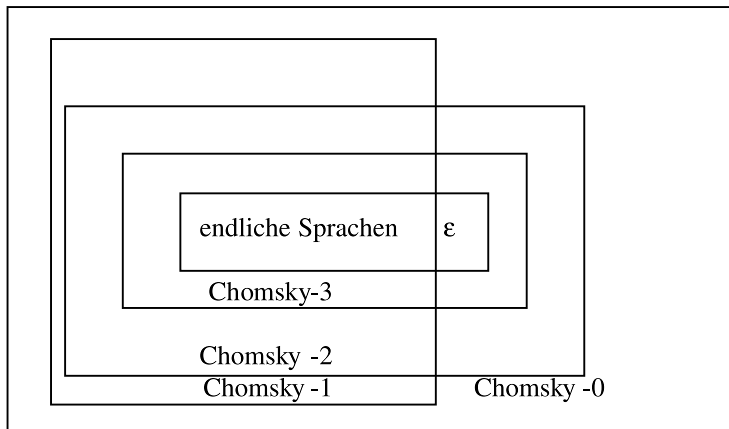
Hieraus ergeben sich beispielsweise folgende ableitbare Sätze:

1. der ball fliegt
2. die heimat winkt
3. das kind schreit
4. ein ball schreit
5. ein heimat fleigt
6. ein kind fliegt

Je nach Art der Regelzuweisung lassen sich Grammatiken in Typen einteilen. Dies geschieht nach dem Gründer der Theorie der formalen Sprachen Noam Chomsky (wesentliche Arbeiten 1955-1965). Man unterscheidet Chomsky-0, Chomsky-1, Chomsky-2, und Chomsky-3 Grammatiken. [Mod 1988, S. 125-126].
Chomsky-0: allgemeine Sprachen
ohne alle Einschränkungen können nichtleere Teilmengen durch andere ersetzt werden.
Chomsky-1: kontext-sensitive Sprachen
Chomsky-2: kontextfreie Sprachen
Chomsky-3: reguläre Sprachen

Beispiel 2 gehört zu Chomsky-2-Grammatiken

Umfang der Sprachklassen



Einige wichtige Eigenschaften formaler Sprachen:
1. Jede endliche Sprache läßt sich durch ein Chomsky-3-Grammatik beschreiben
2. Es gibt unendliche, durch Chomsky-3-Grammatiken beschreibbare Sprachen
(Übung zu Sprache, Aufgabe 1(a))
3. Jede Chomsky-3-Grammatik ist auch vom Typ Chomsky-2.
4. Es gibt Sprachen, die durch Chomsky-2, nicht aber durch Chomsky-3-Grammatiken beschreibbar sind.
(Übung zu Sprache, Aufgabe 1(b,c))
5. Jede ε-freie Chomsky-2-Grammatik (d.h. die keine Regel N → εenthält ist auch vom Typ Chomsky-1.
6. Es gibt Sprachen, die durch Chomsky-1-Grammatiken, nicht aber durch Chomsky2--Grammatiken beschreibbar sind.
(Übung zu Sprache, Aufgabe 1(d))
7. Jede Chomsky-1-Grammatik ist auch vom Typ Chomsky-0
8. Es gibt Sprachen, die durch Chomsky-0, nicht aber durch Chomsky-1-Grammatiken beschreibbar sind.

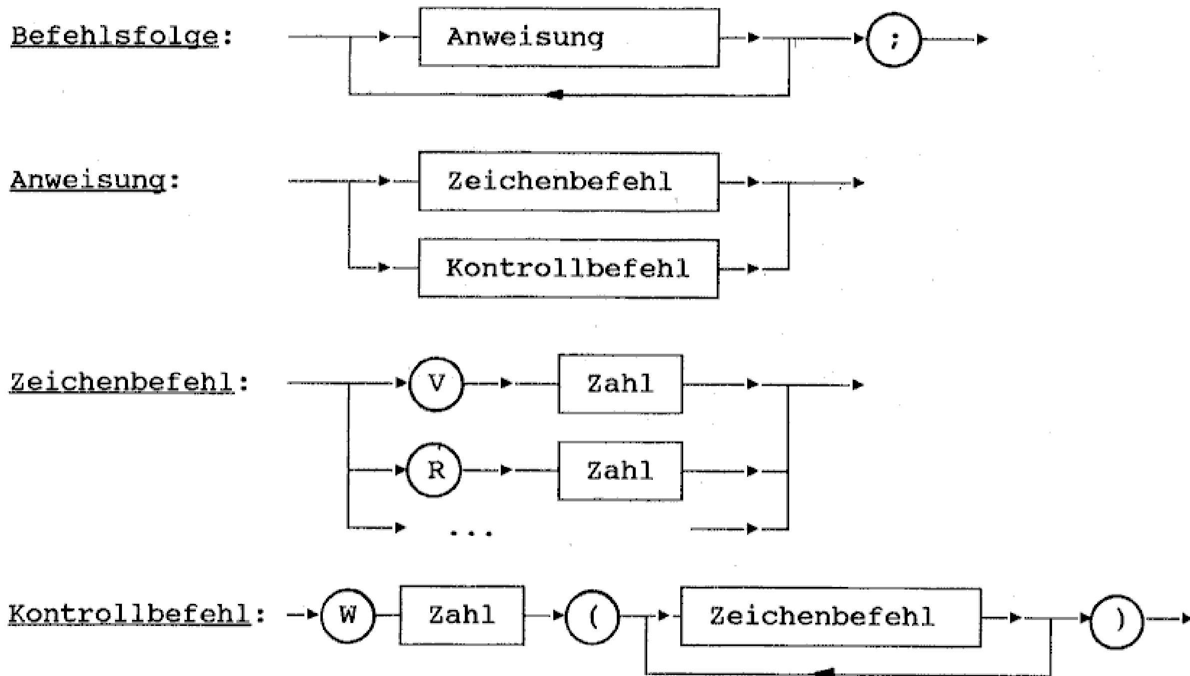
Wichtig für uns sind die Sprachen vom Typ 2 und 3, zu denen zwar keinen lebendigen Sprachen, aber die Programmiersprachen gehören.

Beispiel: LOGO für Arme

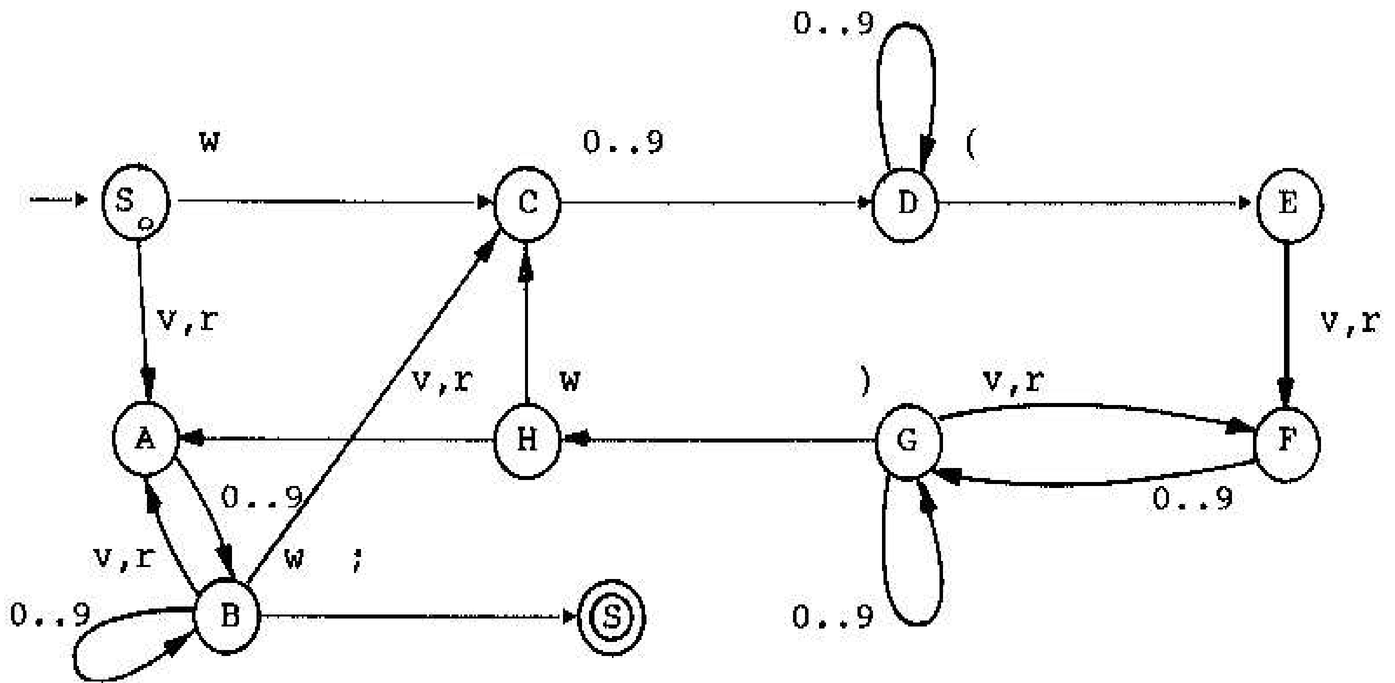
Im folgenden soll beispielhaft dargestellt werden, wie sich einige der genannten Ziele bei der Entwicklung einer Programmiersprache verwirklichen lassen. Das Beispiel stammt aus ([Mod 92] , S. 356 ff) , ist auf der einen Seite noch relativ übersichtlich und damit auch von den Schülern eines Grundkurses nachvollziehbar, bietet die Möglichkeit, im Unterricht nach Leistungsstärke der Schüler zu differenzieren und kann auch noch ausgebaut werden, indem man die Grafiken darstellen läßt.

LOGO für Arme ist eine „Programmiersprache“ ähnlich einer Turtlegraphik. Sie stellt mit wenigen Befehlen ein Werkzeug zur Erstellung von graphischen Elementen zur Verfügung. Damit die theoretischen Modelle überschaubar bleiben, beschränkt man sich zunächst auf Linien und

Vielecke. Dann genügen im Prinzip die beiden Befehle „Vorwärtsgehen“ und „Drehen“ wobei Strichlänge und Winkel durch eine zusätzliche Zahlenangabe festgelegt werden. Es wird davon ausgegangen, daß während des Vorwärtsgehens immer gezeichnet wird. Damit läßt sich diese Sprache durch die folgenden Syntaxdiagramme beschreiben:



Man kann dann, ausgehend von diesen Sprachelementen, einen endlichen Automaten konstruieren, der diese Sprache akzeptiert. Überlegenswert wäre es, ob man nicht zunächst mit noch weniger Sprachelementen diese Aufgabe angeht, um einen etwas einfacheren Automasten zu erhalten. Dies wird abhängen von der Leistungsfähigkeit des jeweiligen Kurses, den erreichten Lernzielen und durchgeführten Übungen aus der 2. Unterrichtseinheit, sowie natürlich auch von der zur Verfügung stehenden Zeit.
Der endliche Automat hätte dann das folgende Aussehen:



Die zugehörige Grammatik gestaltet sich wie folgt:

Nichtterminalsymbole : NT = { A, B, C, D, E, F, G, H, S }
Terminalsymbole: T = { v, r, w, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), ; }
Regelsystem:

S ⟶ B ; | H ;
B ⟶ B0 | ... | B9 | A0 | ... | A9
A ⟶ Bv | Br | Hv | Hr | v | r
H ⟶ G)
G ⟶ G0 | ... | G9 | F0 | ... | F9
F ⟶ Gv | Gr | Ev | Er
E ⟶ D (
D ⟶ D0 | ... | D9 | C0 | ... | C9
C ⟶ Hw | Bw | w

Man hat hier drei unterschiedliche Darstellungsformen , die die selbe Sprache beschreiben. Die Äquivalenz dieser Darstellungsformen läßt sich anhand von Beispielen zeigen. So erzeugt z.B. die Anweisungsfolge w4(v100r90); ein Quadrat. Hier lassen sich auch vielfältige Übungen für die Schüler denken.

Falls im Kurs entsprechende Programmiererfahrung vorhanden ist, lassen sich zu dieser Sprache auch Scanner, Parser und Interpreter z. B. in PASCAL erstellen, so daß letztlich nach Eingabe der Anweisungsfolge diese analysiert, auf richtige Syntax überprüft und anschließend die daraus resultierende geometrische Figur auf den Bildschirm gezeichnet wird. Hier bietet sich dann auch die Möglichkeit, arbeitsteilig zu verfahren oder die Schüler die erforderlichen Programmodule in Form einer Hausarbeit erstellen zu lassen.

Falls dies wegen mangelnder Erfahrung im Programmieren nicht möglich sein sollte, wird man sich auf das Scannen und Parsen von Anweisungsfolgen anhand der oben erwähnten Simulati-

onsumgebung beschränken müssen. Die resultierenden Figuren müssen dann von Hand erstellt werden.

4. Ausblick

Mögliche weitere Inhalte

- Komplexitätsklassen von Problem unterscheiden können
- Probleme zu den Komplexitätsklassen nennen können
- Probleme den Kompexitätsklassen zuordnen können
- Compilerbau als Anwendung von Automaten
- ein zweites Modell der Berechenbarkeit als Alternative zur Turingmaschine
- Church'sche These
- Gödelscher Unvollständigkeitssatz

Komplexitätsklassen

Die Frage nach der Effizienz von Algorithmen ergibt sich bei Problemstellungen, zu denen es verschiedene Lösungsalgorithmen gibt. Eine fundamentale Idee der Informatik im Zusammenhang mit der Bewertung von Algorithmen ist die Komplexität. Programme und Algorithmen, die man in der Praxis verwendet, sollten ein vorgegebenes Problem in möglichst kurzer Zeit lösen und dabei so wenig Speicherplatz wie möglich benötigen. Sie sollten also möglichst effizient arbeiten.
Man untersucht getrennt die Laufzeitkomplexität und die Speicherplatzkomplexität von Algorithmen. Hierbei spielt heute die Speicherplatzkompexität keine große Rolle mehr - Speicherplatz steht in der Regel genug zur Verfügung. Somit ist das wichtigste Kriterium für die Komplexität eines Algorithmus sein Laufzeit. Die Begriffe Laufzeit und Komplexität überträgt man von Algorithmen auf Probleme, indem man als Laufzeit/Komplexität eines Problems die Laufzeit/Komplexität des effizientesten Algorithmus auffaßt, der das Problem löst.

Definition: Die Komplexität eines algorithmisch lösbaren Problems ist definiert als Komplexität eines besten, das heißt effizientesten, Algorithmus unter allen Algorithmen, die dieses Problem lösen.

Als Laufzeit definiert man die Anzahl der Elementaroperationen $T(n)$, die der Algorithmus bei der Eingabe der Größe n im Höchstfall (*worstcase*) ausführen muß. Im allgemeinen interessiert man sich nicht für den genauen Funktionswert von $T(n)$ für jedes Eingabezeichen n, sondern für den qualitativen Verlauf, die Größenordnung. Man sucht also Aussagen in der Form: das Zeitverhalten eines Algorithmus ist proportional zu n (in Zeichen: $T(n)=O(n)$), verhält sich quadratisch, $T(n)=O(n^2)$ oder wie der Logarithmus, $T(n)=O(\log n)$.

Von besonderem Interesse für die Praxis ist der Unterschied zwischen Problemen mit polynomieller Laufzeit, also $T(n)=O(n^p)$ mit $p\in\mathbb{N}$ (siehe Kapitel 1: Berechenbarkeit), und solchen mit nicht-polynomieller Laufzeit, wie z.B. exponentielles Verhalten. Zur genaueren Untersuchung von Problemen mit polynomieller Laufzeit teilt man auch diese in die Komplexitätsklassen **P** und **NP** ein.

Zur Komplexitätsklasse P zählt man alle Probleme, die ein deterministischer Algorithmus in Polynomialzeit löst, NP ist die Menge aller Problem, die ein nichtdeterministischer Algorithmus in Polynomialzeit löst. Da jeder DEA mit polynomiellen Zeitverhalten auch als NEA aufgefaßt werden kann gilt $P \subseteq NP$. Es ist eine offenes Problem in der Informatik ob $P=NP$ oder $P \neq NP$ ist. Beispiele für Probleme mit der Komplexitätsklasse NP ist das Komitee und Erbteilungsproblem aus der ersten Unterrichtsreihe. Da man zu diesen Problem keine effizienten deterministischen Algorithmen angeben kann, vermutet man, daß $P \neq NP$ [Sch 94].

Weitere Berechenbarkeitsbegriffe:

In unserer Darstellung erörtern wir bisher den Begriff der *Turingberechenbarkeit*, unser Algorithmusbegriff wurde also mittels einer Turingmaschine erklärt. Turing: „*Jedes Problem, das überhaupt maschinell lösbar ist, kann von einer Turingmaschine gelöst werden.*“ [Bau93, S. 260] Historisch gesehen, war dieser Weg aber nur einer unter mehreren, die von verschiedenen Theoretikern eingeschlagen wurde. Es sind z. B. zu nennen: [Bau88, S. B17]

- allgemein rekursive Funktionen — 1931-36 — Herbrand, Gödel, Kleene
- Lambda-K-Definierbarkeit — 1936 — Church, Kleene
- Semi-Thue-Systeme — 1914 — A. Thue
- Postsche kanonische Systeme — 1943 — Post
- Markov-Algorithmen — 1954 — Markov
- Registermaschine — 1963 — Shepardson, Sturgis

In der letzten Zeit benutzt man auch häufig die Begriffe *Pascal-Berechenbarkeit* oder *while-Berechenbarkeit*. Letztere wird auch als *Kern von PASCAL* bezeichnet. Hierbei denkt man sich eine Programmiersprache, die aus einfachen Elementen besteht:

- Konstante (Zahlen aus N)
- Variablen: x0, x1, x2, ...
- Operationen: +, -
- Strukturen: Zuweisung := , Trennung ;
- while x<>0 do ... end (eine Variable wird auf Null getestet)

Letztendlich ist es so, daß alle diese Berechenbarkeitsbegriffe gleichwertig sind. Dieser Sachverhalt wird als These von *Church-Turing* formuliert: *Jeder der oben erwähnten exakten Berechenbarkeitsbegriffe stellt all das dar, was wir unter „berechenbar“ oder „algorithmisierbar“ verstehen.*

Erstaunlich ist natürlich auch die Tatsache, daß die Berechenbarkeitsbegriffe von Forschern wie Gödel, Turing, Kleene, u.v.a. in den 30er Jahren entdeckt wurden, in einer Zeit, in der von Computern nicht die Rede war. Sie zeigten damit die Grenzen der mathematisierenden Fähigkeiten des Menschen auf, und zwar mit mathematischen Methoden!

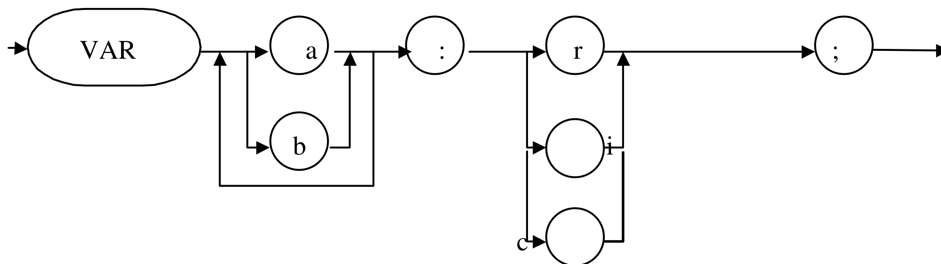
Für die Schule erhebt sich hier die Frage, ob die Behandlung nur eines der Berechenbarkeitsbegriffe genügt. Wer hier in seinem Unterricht noch die Zeit für ein zweites Modell findet, wird sich vielleicht für die PASCAL- oder die while-Berechenbarkeit entscheiden.

Aufgaben aus dem Bereich DEA:

1) Erweitere den Colaautomaten so, daß er auch zwei Markstücke akzeptiert. Die Ausgabe besteht dann aus dem gewünschten Getränk und einer 50Pf-Münze. Es sei hierbei vorausgesetzt, daß der Münzspeicher immer mindestens eines dieser Geldstücke enthält.

2) In Pascal besteht ein Bezeichner aus einem Buchstaben, gefolgt von einer beliebig langen Zeichenkette, die aus Buchstaben, Ziffern oder dem Unterstrich gebildet wird.
Entwickle einen Automaten, der prüft, ob ein Bezeichner korrekt gebildet ist. [Bur88, S. 92]

3) In einer reduzierten P rogrammiersprache werden Variablen nach folgendem Syntaxdiagramm deklariert.



Der auf das Schlüssselwort VAR folgende Bezeichner setzt sich also nur aus den Buchstaben a und b zusammen, es folgt der Doppelpunkt, danach der Typ r, i oder c (dies steht für Real, Integer oder Char). Am Ende wird ein Semikolon gesetzt.
Erstelle einen Automaten, der eine Variablendeklaration auf Korrektheit überprüft.
(Beachte: In der Simulationsumgebung ist das Semikolon als Eingabezeichen nicht erlaubt. Als Ersatz kann das Komma dienen. Ein Leerzeichen zwischen VAR und dem Bezeichner ist nicht vorgesehen.)

4) Eine Mausefalle läßt sich als endlicher Automat auffassen mit der Eingabemenge {Maus kommt, Maus kommt nicht}, der Ausgabemenge {Maus gefangen, Maus nicht gefangen} und der Zustandsmenge {Falle gespannt, Falle nicht gespannt}.
a) Gib die Übergangstabelle und den Zustandsgraphen an.
b) Erweitere die Zustandsmenge um das Vorhandensein von Speck. [Bau81, S. 212]

5) In den Vereinigten Staaten wünscht die *League Against Sexist Speech* ein Programm, das in allen Texten die Zeichenfolge „man“ finden und durch „person“ ersetzen kann. Z. B. soll aus dem „travelling salesman“ eine „travelling salesperson“ gemacht werden.
Entwerfe einen Automaten, der in einem vorgegebenen Text die Silbe „man“ erkennt. [Bau93, S. 216f]

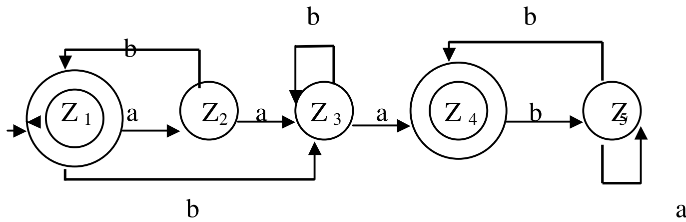
6) Paritätsprüfung (Paritätsbits werden oft als einfaches Mittel zur Fehlererkennung eingesetzt): Gib einen Automaten an, der prüft, ob eine Dualzahl eine ungerade Anzahl von Einsen enthält. [Bau93, S. 222]

7) Gesucht ist ein Automat, der untersucht, ob eine Dualzahl durch die Zahl 4 teilbar ist. Anders formuliert: gesucht ist ein Automat, der erkennt, ob eine Dualzahl auf 100 endet.

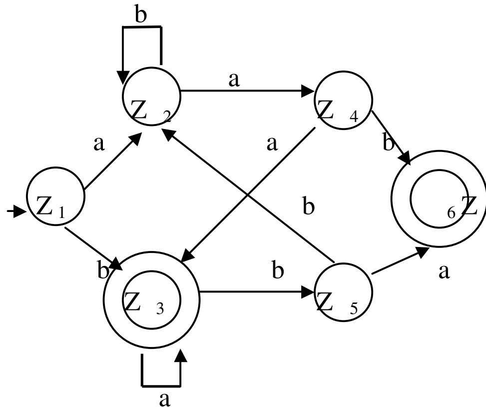
8) Ein Spieler spielt gegen die Bank. Er wirft wiederholt einen Würfel. Er hat das Spiel verloren, sobald er eine gerade Zahl gewürfelt hat, er hat gewonnen, sobald eine 1 und eine 3 (in beliebiger Reihenfolge) gefallen sind. Wenn eine 5 gewürfelt wird, darf er weiterwürfeln.
Beispiele für Gewinnfolgen sind: 1,5,3 oder 5,5,3,3,1
Beispiele für Verlustfolgen sind: 1,5,2 oder 4 oder auch 3,5,6 [InT292, S. 113]
Setze die Spielregeln in einen endlichen Automaten um.

9) Welche der folgenden Worte werden durch den angegebenen Automaten akzeptiert?

(a) ab, aab, baa, aabba,aabbab, baba, bbba, bb, a, b, aaaabbaacaa, aabbaabbaab, ababab



(b) ab, aab, baa,aabba, aabbab, baba, bbba, bb a, b, aaaabbaacaa, aabbaabbaabb, abababa



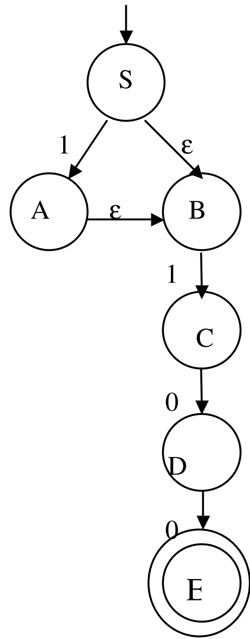
Aufgaben aus dem Bereich NEA:

10) Gegeben sei die nebenstehende Übergangstabelle. Zeichne den zugehörigen Zustandsgraphen und erstelle in der Simulationsumgebung eine entsprechende Übergangstafel. Teste!
Charakterisiere die akzeptierten Worte.
Wird 11001101000 bzw. 110011001000 akzeptiert? Beschreibe das Durchlaufen des Graphen.

	0	1
s	s,1	s
1	2	-
2	3	3
3	4	-
4	4	4

11) Erstelle zu den Aufgaben 5) und 7) jeweils einen nichtdeterministischen Automaten und vergleiche mit dem entsprechenden DEA.

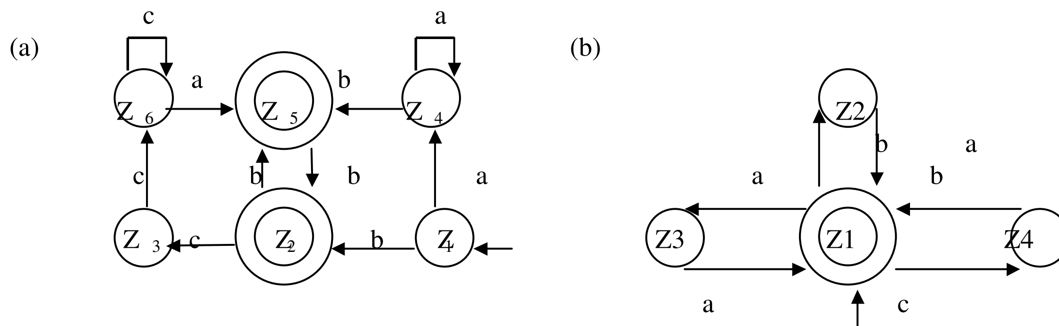
12) Wandle den durch den Graphen gegebenen NEA in einen äquivalenten DEA um. Prüfe dein Ergebnis in der Simulationsumgebung nach.



13) Gib einen NEA an, der die in Pascal zulässigen REAL-Zahlen akzeptiert. Zulässig sind z. B.: 1.4e-5; +3.2e+21; 1.0 [Bur88, S. 92]

Aufgaben zu Automaten und reguläre Sprachen:

14) Geben Sie nun umgekehrt die Sprache an, die durch folgenden Automaten akzeptiert wird.



15) Entwerfen Sie für die folgenden Ausdrücke einen Automaten.
(a) (ac*bc*a) ∪ c(abc)*c
(b) (abb)*abaa*bb*a((abb)*abaa*bb*a))*

16) Eine aus den Symbolen „(“ bzw. „)“ bestehende Eingabe soll überprüft werden, ob die Klammerung nach algebraischen Regeln korrekt ist. Hierbei dürfen nicht mehr als drei Klammern gleichzeitig geöffnet sein.
a) Erstelle eine entsprechenden Automaten.
b) Bearbeite das Problem für maximal fünf gleichzeitig geöffnete Klammern.
c) Begründe, daß kein EA in der Lage ist, daß Problem für beliebig viele Klammern zu lösen.

17) Zu jeder Grammatik, bei der die Produktionsregeln die Form V → aA haben, läßt sich ein EA angeben, der die gleiche Sprache akzeptiert. Erläutere dies anhand des Beispiels:
Gegeben sei die Grammatik mit den Terminalen {a,b}, den Nichtterminalen {S,A,B} und den Produktionen
S → aA, A → aA|bB, B → aA|bB|ε.
Gib den Zustandsgraphen des EA an und vergleiche. [Bau93, S. 223]

Lösungen:

1)
```
a 5 -> 0 1
a 1 -> 0 2
a c -> 0 a
a l -> 0 a
a k -> 0 a

1 5 -> 0 2
1 1 -> 0 3
1 c -> 0 1
1 l -> 0 1
1 k -> 5 a

2 5 -> 0 3
2 1 -> 0 4
2 c -> 0 2
2 l -> 0 2
2 k -> 1 a

3 5 -> 0 4
3 1 -> 1 3
3 c -> C a
3 l -> L a
3 k -> 6 a

4 5 -> 5 4
4 1 -> 1 4
; in den nächsten Zeilen steht g für Getränk + Rück-
geld
4 c -> g a
4 l -> g a
; in der nächsten Zeile be-
deutet 2 die Ausgabe von 2DM
4 k -> 2 a
```

3) ; aus internen Gründen kann das Semikolon nicht als Eingabezeichen
; benutzt werden, es wird deshalb hier durch ein Komma ersetzt!!
; aus Gründen der Vereinfachung wird auf die Eingabe eines Leerzeichens nach
; dem Wort VAR verzichtet.

```
a V -> * 1
a A -> * f
a R -> * f
a a -> * f
a b -> * f
a : -> * f
a r -> * f
a i -> * f
a c -> * f
a , -> * f

1 A -> * 2
1 V -> * f
1 R -> * f
1 a -> * f
1 b -> * f
1 : -> * f
1 r -> * f
1 i -> * f
1 c -> * f
1 , -> * f

2 V -> * f
2 A -> * f
2 R -> * 3
2 a -> * f
2 b -> * f
2 : -> * f
2 r -> * f
2 i -> * f
2 c -> * f
2 , -> * f

3 V -> * 1
3 A -> * f
3 R -> * f
3 a -> * 4
3 b -> * 4
3 : -> * f
3 r -> * f
3 i -> * f
3 c -> * f
3 , -> * f

4 V -> * f
4 A -> * f
4 R -> * f
4 a -> * 4
4 b -> * 4
4 : -> * 5
4 r -> * f
4 i -> * f
4 c -> * f
4 , -> * f

5 V -> * f
5 A -> * f
5 R -> * f
5 a -> * f
5 b -> * f
5 : -> * f
5 r -> * 6
5 i -> * 6
5 c -> * 6
5 , -> * f

6 V -> * f
6 A -> * f
6 R -> * f
6 a -> * f
6 b -> * f
6 : -> * f
6 r -> * f
6 i -> * f
6 c -> * f
6 , -> * e
```

4) Abkürzungen: M: Maus, g: gefangen, N: nicht, F: Falle, G: gespannt, S steht für Speck

	Maus kommt	M kommt nicht
FG	Mg/FNG	MNg/FG
FNG	MNg/FNG	MNg/FNG

	Maus kommt	M kommt nicht
FG	Mg/FNG	MNg/FG
FNG	MNg/FNG	MNg/FNG
FGS	Mg/FNG	MNg/FGS
FNGS	MNg/FNG	MNg/FNGS

7)

	0	1
S	F	1
1	2	3
2	E	1
3	2	3
E	1	1

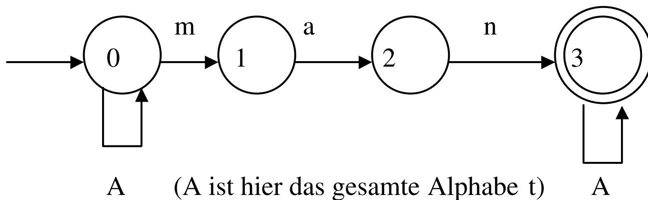
8) Nach Erreichen von sG (Gewinn) bzw. sV (Verlust) wird nicht weitergewürfelt.

	2,4,6	1	3	5
s0	sV	s1	s2	s0
s1	sV	s1	sG	s1
s2	sV	sG	s2	s2
sG	sG	sG	sG	sG
sV	sV	sV	sV	sV

9) Folgende Ausdrücke werden akzeptiert:
(a) ab, baa, aabba, bbba, ababab
(b) aab, baa, baba, b

10)
```
a 1 -> * a
1 0 -> * 2
2 0 -> * 3
2 1 -> * 3
3 0 -> * e
e 0 -> * e
e 1 -> * e
```
Es werden Worte akzeptiert, die als Teilwort entweder 0000 oder 0010 enthalten. Beim Durchlaufen, das in einem Zielzustand enden soll, muß Backtracking einsetzen. Zeichenweises Abarbeiten des ersten Wortes führt zu folgenden Zustä nden: ss123, hier muß zurückgesetzt werden, eine Alternative im Durchlaufen ist nur bei Eingabe der ersten 0 möglich: sss1. Auch hier endet der Versuch, deshalb: ssssss1. An dieser Stelle spätestens erkennt man, daß der Übergang von s nach 2 nur über die Eingabe von zwei aufeinanderfolgenden Nullen möglich wird, das erste Wort nicht akzeptiert.
Beim zweiten Wort ergibt sich die Folge von Zuständen als: ssssss123eee

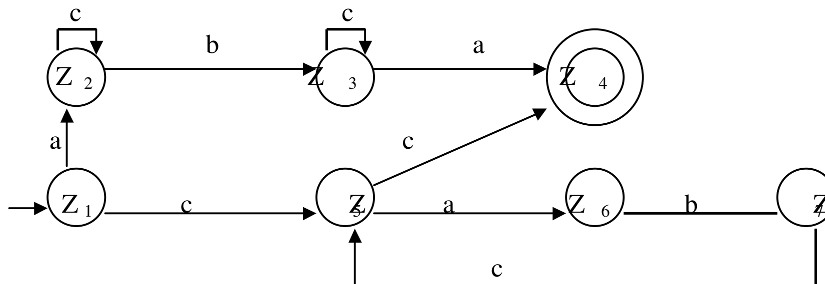
11)



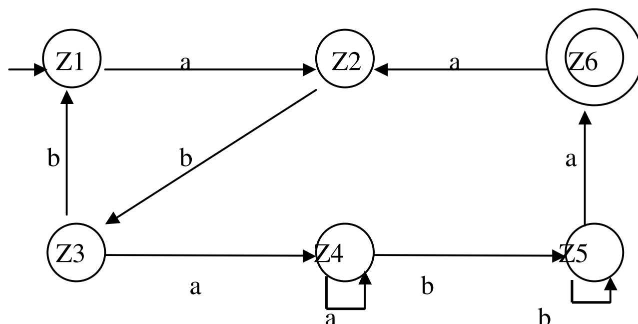
s. Aufg. 12.

14) Der reguläre Ausdruck zum gegebenen Automaten sieht folgendermaßen aus:
(a) aa*b(bb∪bacc*a)*∪b(bb∪acc*ab)*∪aa*b(bb∪acc*ab)*∪bb(bb∪bacc*a)*
(b) (aa∪ab∪cb)*

15)
(a)



(b)



16) Man erkennt an diesem Beispiel, daß die Erweiterung auf z. B. vier Klammern es nötig macht, auch einen weiteren Zustand einzuführen. Es ist also nicht möglich, mit einem endlichen Automaten (**endlich** viele Zustände) beliebige Klammerungen zu untersuchen. Um dies leisten zu können, muß der Automat über ein „besseres“ Gedächtnis verfügen.

	(	)
s	1	f
1	2	s
2	e	1
e	f	2

17) Baumann gibt hier eine Konstruktionsvorschrift an, die es erlaubt, zu jeder rechtslinearen (also auch regulären) Grammatik einen endlichen Automaten zu konstruieren, der die gleiche Sprache akzeptiert. In [Bau81, S. 257] gibt er eine hierzu analoge Vorschrift für linkslineare Grammatiken an.

rechtslineare G.	Endl. Automat
A -> aB	A —a→ B
A -> a	A —a→ z_e (Endzustand)
S	z_0

linkslineare G.	endl. Automat
A -> Ba	B —a→ A
A -> a	z_0 —a→ A
S	z_e

Übungen zu Sprache/Grammatik

Aufgabe 1:
Gegeben sind eine Reihe von Sprachdefinitonen über dem Terminal Alphabet A = {a,b,c,d}. Sie sollen (mit Begründung) den Typ der Sprache und die entsprechende Grammatik zur Generierung der Sprache definieren.
(a) $L_1 = \{ab^nc \mid n \geq 1\}$
(b) $L_2 = \{a^nb^n \mid n \geq 1\}$
(c) $L_3 = \{a^nb^n \mid n \geq 0$
(d) $L_4 = \{a^nb^na^n \mid n \geq 1\}$
(e) $L_5 = \{a^nb^nc^n \mid n \geq 0\}$
(f) $L_6 = \{wcw^R \mid w \in (a \mid b)^*\}$ und w^R ist das Spiegelbild von w.
(g) $L_7 = \{a^nb^mc^md^n \mid n, m \geq 1\}$

Aufgabe 2:
In einer Grundschul-Fibel steht der folgende Satz:
HANS REDET NICHT MIT ANNA. SIE SEHEN SICH NICHT AN. WAS IST NUR LOS? KEINER LACHT. KEINER REDET EIN WORT.
(a) Nennen Sie eine Grammatik, die diese Sätze erzeugen kann.
(b) Entwickeln Sie einen Automaten, der diese Sätze akzeptiert.
(c) Zeichnen Sie einen Syntaxbaum für die angegebenen Sätze. [MOD 1988, S. 134]

Aufgabe 3:
Entwickeln Sie einen Automaten und die zugehörige Grammatik für die Sprache aus natürlichen Dezimalzahlen,
(a) die durch 2 (3, 5, 100) teilbar sind.
(b) die ungerade sind. [MOD 1988, S. 134]

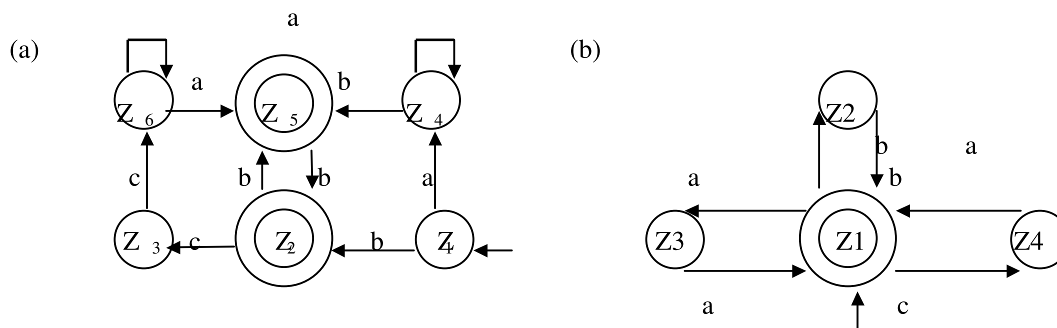
Aufgabe 4:
Geben Sie zu der gegebenen Grammatik die erzeugte Sprache und den Typ der Sprache in der Chomsky-Hierachiean. Gegeben ist die folgende Grammatik
$G = (N, T, S, R)$ mit $N = \{S, A\}$, $T = \{0,1\}$, $S = S$, $R = \{ S \rightarrow 0A1, 0A \rightarrow 00A1 \mid \varepsilon\}$
(a) Welche Sprache L(G) wird durch G erzeugt?
(b) Von welchem Typ ist L(G)?

(c) Schreiben Sie einen mindestens 4-stufigen Ableitungsbaum auf.

Aufgabe5:
Geben Sie für die folgenden regulären Ausdrücke eine Grammatik an, einschließlich der entsprechenden Mengendefinitionen mit A = {a,b}.
(a) A*
(b) (a∪b)*a(a∪b*)
(c) a*b*

Aufgabe 6:
Gegeben sind die Mengen: N = {S, E, F, T}, T = {a,b,c,0,1,...,9,+,-,*,/,(,)}, S = E. Definieren Sie nun jeweils Sprachen für die Chomsky-Klassen 0-3 mit den dazu gehörigen Produktionen (Regeln).
(a) Chomsky-1 Sprache, die nicht Chomsky-2 ist
(b) Chomsky-2 Sprache, die nicht Chomsky-3 ist
(c) Chomsky-3 Sprache, die nicht endlich ist
(d) eine endliche Sprache.

Aufgabe 7:
Geben Sie die Grammatik an, die durch folgenden endlichen Automaten akzeptiert wird.

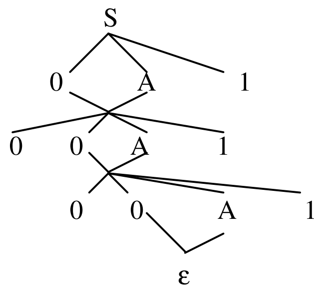


Lösungen zu den Aufgaben:

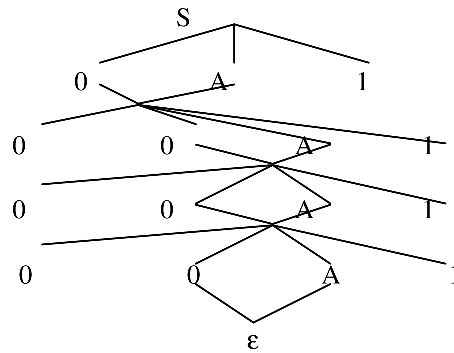
Aufgabe 1:
(a)
G = ({S, X}, {a,b,c}, S, {S → Xc, X → ab, X → Xb})
ist eine Chomsky-3-Grammatik.
(b)
G = ({S}, {a,b,c}, S, {S → ab, S → aSb})
ist eine Chomsky-2- aber nicht eine Chomsky-3 Grammatik
(c)
G = ({S}, {a,b,c}, S, {S → ε, S → aSb})
ist eine Chomsky-2-Grammatik
(d)
G = ({S,X,Y}, {a,b}, S, {S → aS, S → aY, YX → bYa, aX → Xa, Y → ba})
ist eine Chomsky-1 aber nicht eine Chomsky-2-Grammatik
(e)
G = ({S,X,Y}, {a,b,c}, S, {S → aSX, S → aY, S → ε, YX → bYc, cX → Xc, Y → bc})
ist eine Chomsky-0-Grammatik
(f)
G = ({S,X,Y,C}, {a,b,c}, S, {S → C , S → XCX, S → YCY, X → a, Y → b, C → XCX, C → YCY, C → c})
ist eine Chomsky-2-Grammatik, keine Chomsky-3-Grammatik
(g)
G = ({S,X,Y}, {a,b,c,d}, S, {S → X, X → aXd, X → Y, Y → bc, Y → bYc})
ist eine Chomsky-2-Grammatik

Aufgabe 4:
(a) L(G) = {1, 011, 00111, 0001111, ...}

(b) Die Grammatik ist vom Typ Chomsky-0
(c) Zwei Beispiele für Ableitungsbäume:



Entspricht: 00111



Entspricht: 0001111

Aufgabe 5:
Alle gegebenen Ausdrücke sind regulär, deshalb ist nach einer Chomsky-3-Grammatik zu suchen.
(a)
G = ({S,N}, {a,b}, S, {S → N, N → aN, N → bN, N → ε})
(b)
G = ({S,X,Y,Z}, {a,b}, S, {S → X, X → aX, X → bX, X → Y, Y → aa, Y → aZ, Z → bZ, Z → ε})
(c)
G = ({S,A,B}, {a,b}, S, {S → A, A → aA, A → B, A → ε, B →bB, B → ε})

Aufgabe 6:
(a) Chomsky-1-Grammatik, die nicht Chomsky-2 ist:
L(G) = $\{0^n1^n0^n \mid n \geq 1\}$
G = ({S,F,T}, {0,1}, S, { S → 0SF, S → 0T, TF → 1T0, 0F → F0, T → 10})
(b) Chomsky-2-Grammatik, die nicht Chomsky-3 ist:
G = ({S,F,T}, {0,2,5,+,(,),*}, S, { S → S*F|5, F → (T), T → 1+2})
(c) Chomsky-3, die nicht endlich ist:
G = ({S,F}, {0,1,2,3,4,5,6,7,8,9,+}, S, { S → 0+S|1+S|2+S|3+S|4+S|5+S|6+S|7+S|8+S|9+S
, S → F, F → 0|1|2|3|4|5|6|7|8|9})
(d) eine endliche Sprache:
G = ({S,F,T}, {0,1,2,3,4,5,6}, S, { S → 1*F|2*F, F → 3+T|4+T, T → 5|6})
L(G)= {1*3+5, 1*3+6, 1*4+5, 1*4+6, 2*3+5, 2*3+6, 2*4+5, 2*4+6}

Aufgabe 7:
(a)
S ist Anfangszustand, U und V sind Endzustände
G = ({S,T,U,V,W,X}, {a,b}, S, {(S,a) → T, (S,b) → V, (T,a) → T, (T,b) → U, (U,b) → V,
(V,a) → X, (V,b) → U, (X,c) → W, (W,c) → W, (W,a) → U})
(b)
A ist Anfangs- und Endzustand
G = ({A,B,C,D}, {a,b,c}, S = A, {(A,a) → D, (A,b) → C, (A,c) → B, (B,b) → A, (C,c) → A, (D,a) → A})

Literatur

[Bau 88] Bauer, Manfred et al. (1988), *Theoretische Informatik, Algorithmen und ihre prinzipiellen Grenzen:* Hrsg. Landesinstitut für Erziehung und Unterricht Stuttgart.

[Bau 84] Baumann, Rüdeger (1984), *Informatik mit Pascal*: Klett-Verlag, Stuttgart.

[Bau 90] Baumann, Rüdeger (1990), *Didaktik der Informatik*: Klett-Verlag, Stuttgart.

[Bau 93] Baumann, Rüdeger (1993), *Informatik für die Sekundarstufe II,* Band 2, Klett-Verlag, Stuttgart.

[Bur 88] Burkert, J., Grieser, Dr. H., Postel, H. (1988) *Informatik heute*: Algorithmen und Datenstrukturen - Band 2: Schrödel Verlag, Hannover.

[Go/Li 86] Goldschlager, L., Lister, A. (1986), *Informatik, eine moderne Einführung:* Carl Hanser-Verlag, München.

[Hop 90] Hopcroft, J.E., Ullman, J.D. (1990), *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*: Addison-Wessley, Bonn.

[Jae 86] Jaenisch, Prof. (1986), *Automatentheorie*: Lehrerweiterbildung Informatik, HILF/HIBS.

[KSP 94] Kursstrukturpläne Gymnasiale Oberstufe (1994), *Informatik*: Verlag Moritz Diesterweg, Frankfurt.

[Mod 88] Modrow, Eckart (1988), *Automaten, Schaltwerke, Sprachen*: Dümmler-Verlag, Bonn.

[Mod 92] Modrow, Eckart (1992), *Zur Didaktik des Informatikunterrichts*: Band 2, Dümmler-Verlag, Bonn.

[Poh 92] Pohlmann, Dietrich (1992), *Klausuraufgaben Informatik*: Sekundarstufe II, Band 2, Aufgaben/Lösungen: Dümmler-Verlag, Bonn.

[Sch 94] Schwill, Andreas (1994), *Praktisch unlösbare Probleme*: LOGIN 14, Heft 4.