ÜBER BERECHENBARKEIT
EINE ZUSAMMENFASSUNG

T. Hürlimann

Working Paper No. 213

(Letzte Änderung: März 1995)

Juni 1993

Institut für Informatik, Regina Mundi

Universität Freiburg

CH-1700 Freiburg / Schweiz

Bitnet: HURLIMANN@CFRUNI51

Tel: (41) 37 21 95 60 Fax: 37 21 96 70

Über Berechenbarkeit
eine Zusammenfassung

Tony Hürlimann, Dr. rer. pol.

Key-words: Computation, Turing-Church, Turingmachine

Abstract: This paper gives a brief overview of the classical theory of computation: theory of automata (finite automaton, stack automaton and Turing machine), the theory of language (Chomsky's hierarchy), recursively enumerable sets and partial recursive functions. An outline of the proofs for the halting problem and for Gödel's theorem of incompleteness is proposed. A short note about intelligence concludes the paper.

Stichworte: Berechenbarkeit, Church-These, Turing-Maschine

Zusammenfassung: Dieses Paper soll einen Überblick über die klassischen Begriffe der Berechnungstheorie aus Sicht der Informatik geben: Automatentheorie, Sprachtheorie, rekursive Mengen und Funktionen. Etwas ausführlicher wurde das Halteproblem behandelt. Am Schluss soll eine kurze Beweisskizze des Gödelschen Satzes, wie sie von Nagel und Newman [1958] präsentiert wurde, vorgestellt werden. Dieser Satz ist von so grosser Bedeutung für die theoretische Informatik, dass zumidest seine Idee für jeden Informatiker zur 'Allgemeinbildung' gehören sollte. Eine kurze Notitz über Intelligenz beschliesst das Paper.

"....there is no such thing as an unsolvable problem."
(Hilbert 1930, zitiert nach Hodges S.92)

Was macht ein Barbier, der allen die Bärte rasiert,
die sich selber nicht rasieren, mit *seinem* Bart?

"Die mathematische Präzisierung des Berechenbarkeitsbegriffs und die Erkenntnis der Grenzen des algorithmisch Machbaren gehören zu den wichtigsten intellektuellen Leistungen des 20. Jahrhunderts." (Wagner, S 2).

1. EINLEITUNG

Ein wichtige Aufgabe der theoretischen Informatik ist die Klärung der Begriffe der **Berechenbarkeit** und des **Algorithmus**. Intuitiv ist ein Algorithmus ein endliches Verfahren, um eine Aufgabe mechanisch zu lösen. Man sagt, dass ein Problem **unentscheidbar** ist, wenn es nachweislich keinen Algorithmus für seine Lösung gibt. Ein Problem wird so formuliert, dass zu seiner Beantwortung 'ja' oder 'nein' genügt. Das Problem 'Ist 29 eine Primzahl?' ist ein typisches Problem, wofür es einen einfachen Algorithmus gibt, der 'ja' ausgibt, wenn 29 eine Primzahl ist und 'nein', wenn 29 keine Primzahl ist. Hingegen ist das Problem 'hält das Programm X?' unentscheidbar, da man zeigen kann, dass dafür kein Algorithmus geschrieben werden kann, der für alle Programme X die richtige Antwort gibt. Bei einer Funktion y=f(x) sagt man, dass sie **nicht berechenbar** ist, wenn es keinen Algorithmus gibt, der für jedes Element x der Definitionsmenge D (mit x□D) den Wert y berechnen kann, obwohl die Funktion f wohldefiniert ist. Der Begriff der Berechenbarkeit ist also für die Funktion, was der Begriff Entscheidbarkeit für ein Ja-Nein-Problem ist.
Es ist nicht selbstverständlich, dass es unentscheidbare Probleme gibt, wie der einleitende Satz von Hilbert zeigt. Dass es unentscheidbare Probleme gibt, hat erstmals Gödel 1931 mit seinem Unvollständigkeitstheorem zeigen können. In diesem Artikel soll die Menge der berechenbaren Funktionen, die wir mit der Menge der **partiell-rekursiven Funktionen** identifizieren werden, etwas näher beleuchtet werden. Die Menge der entscheibaren Problem wird mit der Menge der **rekursiv aufzählbaren Mengen** identifiziert. Das Studium dieser Mengen gehört zur **Berechenbarkeitstheorie**. Sie befasst sich damit, für welche Probleme es ein (mechanisches) Entscheidungsverfahren gibt.

Die **Komplexitätstheorie** hingegen stellt die Frage nach der praktischen Durchführbarkeit von Lösungen: Gesetzt der Fall ein Algorithmus existiert für ein gegebenes Problem, wie lange dauert das Auffinden der Lösung und wieviel Speicherplatz wird dabei verbraucht? Dabei stellt sich heraus, dass es viele Probleme gibt, für welche zwar ein Algorithmus bekannt ist, der jedoch

keinen praktischen Wert hat, da dessen Durchführbarkeit selbst mit schnellsten Computern viel zu lange dauern würde oder zuviel Speicher beanspruchen würde.

Beispiel für ein einfaches Problem: finde einen Algorithmus für folgende Aufgaben (Stetter S. 5):

a) Alle natürlichen Zahlen von 1 bis 10^6 ausdrucken.
b) Alle natürlichen Zahlen von 1 bis 10^{11} ausdrucken.
c) Alle natürlichen Zahlen von 1 bis 10^{100} ausdrucken.
d) Alle natürlichen Zahlen ausdrucken.
e) Alle reellen Zahlen ausdrucken.

Ein Algorithmus in einer Pascal-ähnlichen Hochsprache für a) könnte folgendermassen formuliert werden:

```
Schritt 1: z:=1
Schritt 2: Drucke z
Schritt 3: z:=z+1
Schritt 4: Falls z ≤ 10^6, mache bei Schritt 2 weiter
Schritt 5: Stoppe.
```

Mit einer Druckerleistung von 1000 Zahlen pro Minute ist die Aufgabe a) in 17 Stunden durchführbar.
Für die Aufgabe b) muss die vierte Zeile ersetzt werden durch

```
Schritt 4: Falls z ≤ 10^11, mache bei Schritt 2 weiter
```

Für dieselbe Druckerleistung bräuchte man dazu immerhin 190 Jahre! Bei einer 1000-fachen Leistungssteigerung könnte man die Aufgabe in 2 Monaten bewältigen. Diese Aufgabe kann also in absehbarer Zeit (wenn wir vom Papierverbrauch absehen!) noch gelöst werden.
Für Aufgabe c) lautet die vierte Zeile

```
Schritt 4: Falls z ≤ 10^100, mache bei Schritt 2 weiter
```

10^{91} Jahre wäre nötig (das Alter des Weltall beträgt gerade $2*10^{10}$ Jahre). Der Platz zum Drucken der Ergebnisse wäre nicht einmal verfügbar: Das All besteht aus wesentlich weniger als 10^{100} Atomen. Obwohl theoretisch machbar, sprengt diese Aufgabe jeden *praktischen* Rahmen.
Für Aufgabe d) ersetzt man die vierte Zeile durch

```
Schritt 4: mache bei Schritt 2 weiter
```

Die Aufgabe wird durch den Algorithmus zwar korrekt beschrieben, aber man hat jetzt einen Algorithmus, der *nicht hält,* da die Menge der natürlichen Zahlen unendlich ist. Streng genommen kann hier nicht mehr von einem Algorithmus gesprochen werden, da ein wesentliches Element des Algorithmus seine Endlichkeit ist. Für Algorithmen, die halten oder nicht halten, wird das Wort

Prozedur verwendet werden.
Für die Aufgabe e) kann der vorgeschlagene Algorithmus nicht verwendet werden. *Man kann sogar zeigen, dass für dieses Problem kein Algorithmus - nicht einmal einer, der nie hält - angegeben werden kann.* Um dies zu beweisen, verwendet man eine in der Berechnungstheorie typische Argumentation: Man nimmt an, dass ein solches Verfahren existiere. Daraus wird ein Widerspruch konstruiert, woraus zu schliessen ist, dass die Annahme (die Existenz des Verfahrens) falsch war.
Angenommen also, die obige Aufgabe e) habe einen Algorithmus. Dann wird der Algorithmus z.B. die Zahlen zwischen 0 und 1 in einer bestimmten Reihenfolge ausgeben. Angenommen, die in dieser Reihenfolge i-te ausgegebene Zahl sei die Zahl z_i, welche aus den Ziffern $z_{i1}z_{i2}z_{i3}z_{i4}$.... hinter dem Komma besteht. Dann kann die Folge der ausgegebenen Zahlen folgendermassen beschrieben werden:

$z_1 = 0.z_{11}z_{12}z_{13}z_{14}$....
$z_2 = 0.z_{21}z_{22}z_{23}z_{24}$....
$z_3 = 0.z_{31}z_{32}z_{33}z_{34}$....
.....

Man betrache nun eine reelle Zahl $x= 0.x_1x_2x_3x_4$.... zwischen 0 und 1, die so konstruiert ist, dass jede k-te Ziffer x_k von z_{kk} verschieden ist. Diese Zahl kann aber in der ausgedruckten Liste z_1, z_2, z_3, ... nicht vorkommen, da sie ja mindestens in einer Ziffer verschieden von jeder in dieser Liste vorkommenden Zahl ist. Daher können in der Liste nicht alle reelen Zahlen zwischen 0 und 1 vorhanden sein, was ein Widerspruch zur obigen Annahme ist, dass alle Zahlen zwischen 0 und 1 in einer Reihenfolge ausgegeben werden. Es kann also keine Liste von reelen Zahlen geben und damit auch keinen Algorithmus, welcher diese ausdruckt.
Die Argumentation basiert natürlich auf dem berühmten **Diagonalisierungsverfahren** von Cantor, der in einen strengen Sinne nachwies, dass es nicht abzählbare Mengen gibt. In der theoretischen Informatik wird eine Reihe wichtiger Sätze mit Hilfe der Diagonalisierung bewiesen.

Damit ist die Frage bereits beantwortet, ob es unlösbare Probleme gibt. Dies beruht ganz einfach auf der Tatsache, dass es viel mehr Probleme gibt als Lösungen! Diese Aussage wird noch weiter unten präzisiert werden.
Eine Frage schliesst sich sofort an: welche Probleme sind lösbar und welche nicht? Die Antwort auf diese Frage erspart uns das Suchen eines Algorithmus'

für ein Problem, das keinen hat!

In Kapitel 2 wird eine kurze Zusammenstellung der verwendeten mathematischen Begriffe gegeben. Kapitel 3 behandelt die verschiedenen Maschinenmodelle und gibt eine kurze Zusammenfassung der verschiedenen Automaten. Kapitel 4 behandelt die dazu gehörenden formalen Sprachen (Chomsky-Hierarchie). Im Kapitel 5 wird die Hierarchie der Berechenbarkeit aus Sicht der Mengentheorie und berechenbaren Funktionen betrachtet. Kapitel 6 gibt zwei Beispiele von unentscheidbaren Problemen: Das Halteproblem und das Wortproblem. Kapitel 7 gibt die Beweisskizze nach Newman & Nagel wieder.

2. MATHEMATISCHE BEGRIFFE

Im Text wird folgendes, mathematisches Rüstzeug vorausgesetzt.

1. MENGEN UND RELATIONEN

Mengen werden definiert, indem ihre Elemente explizit angegeben werden:

$M = \{a,b,c,d\}$

oder indem eine Vorschrift für ihre Konstruktion angegeben wird:

$P = \{x \square \mathrm{N} \square\ x<10\ \}$

$Q = \{\ x \& x$ ist Primzahl $\}$

$R = \{\ x \square N \& x$ ist gerade $\}$

Die Mengenoperationen $\square, \square, \approx, \leftrightarrow, \wp, \prod, \aleph, \infty$ werden als bekannt vorausgesetzt. Die leere Menge wird mit dem Symbol Ø oder {} bezeichnet. Endliche Mengen enthalten eine endliche Anzahl von Elementen. So sind die oben definierte Mengen *M* und *P* beispielsweise endlich. Unendliche Mengen enthalten eine unendliche Zahl von Elementen. Ist M eine Menge, so nennt man die Anzahl der Elemente ihre Kardinalität, dies wird mit $\square M \subseteq$ abgekürzt. **Abzählbare Mengen** sind unendliche Mengen, die aufgelistet werden können. Die Menge der ganzen Zahlen oder der rationalen Zahlen sind abzählbar. Unendliche Mengen, die nicht abzählbar sind, heissen **überabzählbare Mengen**; die Menge der reelen Zahlen, z.B. ist überabzählbar, ebenso die Menge der Mengen der rationalen Zahlen. Ist M eine Menge, so ist 2^M ihre Potenzmenge. Sie besteht aus allen Teilmengen von M. Ein **Tupel** ist eine geordnete Sequenz von Elementen, z.B. (*a,b,c,d*), Eine **Relation** ist eine Menge von Tupeln und wird geschrieben als R:X∅Y, oder als R∏X∞Y, oder als *y*=R(*x*). Eine Relation kann somit auch als Untermenge eines Kartesischen

Produktes verstanden werden. Eine Relation heisst bijektiv, wenn jedem *x* genau ein Wert *y*, surjektiv, wenn jedem *y* höchstens ein *x*, und injektiv, wenn jedem *x* höchstens ein *y* zugeordent ist.

Eine **Funktion** *y*=f(*x*) ist eine Relation, wobei *x* ein Element aus einem Definitionsbereich D und *y* ein Element aus einem Wertebereich W ist und für jedes *x* höchstens ein Element *y* zugeordnet ist. Eine Funktion ist also eine injektive Relation.

Eine Relation R

$R \subseteq A \times A$ ist reflexiv, wenn $(\forall a)(a,a) \in R$

$R \subseteq A \times A$ ist symmetrisch, wenn $(a,b) \in R \leftrightarrow (b,a) \in R$

$R \subseteq A \times A$ ist antisymmetrisch, wenn $a \neq b \wedge (a,b) \in R \rightarrow (b,a) \notin R$

$R \subseteq A \times A$ ist transitiv, wenn $(a,b) \in R \wedge (b,c) \in R \rightarrow (a,c) \in R$

Eine Kette in R ist eine Sequenz $(a_1, a_2, \ldots, a_n)$ sodass $(a_i, a_{i+1}) \in R$ mit $i = \{1,\ldots, n-1\}$. Die Kette ist ein Kreis, wenn $a_1 = a_n$. Der Kreis ist nicht trivial, wenn n>1.

Eine **Äquivalenzrelation** ist eine Relation die reflexiv, symmetrisch und transitiv ist.

Theorem 1: Eine Äquivalenzrelation R auf die Menge M partitioniert die Menge M in disjunkte Untermengen, genannt Äquivalenzklassen. (Beweis in Hopcroft S. 12).

Eine **Ordnungsrelation** ist eine Relation, welche reflexiv ist und keinen nicht trivialen Kreis enthält.

Die **transitive Hülle** R^+ einer Relation R ist definiert als

(1) Falls (*a*,*b*) in R ist, so ist auch (*a*,*b*) in R^+.

(2) Falls (*a*,*b*) in R^+ und (*b*,*c*) in R ist, dann ist (*a*,*c*) in R^+.

(3) Nichts ist in R^+, ausser es folgt aus (1) oder (2).

Die **Halbgruppe** eine nicht-leere Menge M zusammen mit einer Relation R∏M∞M, welche die beiden folgenden Eigenschaften erfüllt

(1) Abgeschlossenheit: Für alle *a*,*b*□M gilt auch *a*R*b*□M.

(2) Assoziativität: Für alle *a*,*b*,*c*□M gilt (*a*R*b*)R*c* = *a*R(*b*R*c*) = *a*R*b*R*c*.

2. VIER BEWEISTECHNIKEN

Beweis durch Widerspruch (Reductio ad absurdum)

Das Wesentliche bei diesem Beweisverfahren ist, die Annahme zu treffen, die widerlegt werden soll, und daraus einen Widerspruch abzuleiten (reductio ad absurdum).

Klassisches Beispiel: Zeige, dass $\sqrt{2}$ keine rationale Zahl ist.

Wir nehmen an, dass $\sqrt{2}$ eine rationale Zahl ist, dann kann man schreiben:

$$\sqrt{2} = n/m$$

wobei n und m zwei ganze, teilerfremde Zahlen sind. Durch Umformen erhalten wir die Formel

$$2m^2 = n^2$$

Das bedeutet aber, dass n^2 gerade sein muss, was impliziert, dass auch n gerade sein muss, sodass wir n durch $2k$ ersetzen können, wobei k wieder eine ganze Zahl ist:

$$2m^2 = 4k^2 \quad \text{und damit} \quad m^2 = 2k^2$$

Daraus sieht man aber, dass auch m gerade sein muss. Aber dies widerspricht unserer Annahme, dass n und m keinen gemeinsamen Faktoren haben, sonst könnte man ja n/m kürzen. Daher gibt es keine ganze Zahlen m und n, derart dass $\sqrt{2} = n/m$ ist. (Was zu zeigen war).

Die Induktion

In der Praxis wird die Induktion für den Beweis des folgenden Satzes eingesetzt:

"Für alle natürlichen Zahlen gilt die Eigenschaft P(n)".

Der Beweis wird in folgender Form angewandt:

1) Basissatz: Die Eigenschaft gilt für n=0, (P(0) ist wahr).
2) Induktionshypothese: Wenn P(n) gilt, dann gilt auch P(n+1).
3) Induktionsschritt: Aus 1) und 2) folgt, dass auch P(n+1) gilt.

Beispiel: Zeige durch Induktion, dass ein binärer Baum der Höhe n, maximal 2^n Blätter hat. Es ist also zu zeigen, dass $B(n) \leq 2^n$, wenn B(x) die Anzahl Blätter des Baumes mit der Höhe x bezeichnet.

Basissatz: Es gilt: $B(0)=1=2^0$, da ein Baum der Höhe 0 nur die Wurzel enthält.

Induktionsannahme: $B(i) \leq 2^i$ für alle i=0, 1, 2, 3, ..., n

Induktionsschritt: Um von einem Baum der Höhe n zu einem Baum der Höhe n+1 zu gelangen, können wir für jedes Blatt höchstens zwei Äste mit je einem Blatt hinzufügen. Daher haben wir: Wenn eine Baum der Höhe n insgesamt B(n) Blättern hat, so hat ein Baum der Höhe n+1 höchstens 2•B(n) Blätter. Damit haben wir $B(n+1) \leq 2 \bullet B(n) \leq 2 \bullet 2^n = 2^{n+1}$. Ist also $B(n) \leq 2^n$, so ist $B(n+1) \leq 2^{n+1}$. Damit haben wir den Satz bewiesen.

(Um dies noch besser zu veranschaulichen, können wir der Reihe nach n=0, dann n=1, dann n=2, usw. einsetzen).

Das Pigeonhole Prinzip

Theorem 2: Wenn A und B zwei nicht leere, endliche Mengen mit $|A| > |B|$ sind, dann gibt es keine bijektive (bzw. injektive) Funktion zwischen A und B.

Mit andern Worten: Wenn n Tauben und m Taubenhäuschen (mit $n>m$) gegeben ist, dann müssen in einem Häuschen mindestens 2 Tauben sein. Das Prinzip scheint so trivial, dass es schwer ist sich vorzustellen, dass es eine wichtige Bedeutung in der Berechnungstheorie hat. Trotzdem ist dieses Prinzip z.B. im Zusammenhang mit dem Pumping-Lemma sehr nützlich, um zu beweisen, dass eine Sprache nicht in einer Sprachenfamilie ausdrückbar ist (siehe Schöning p. 39 als Beispiel).
Das Pigeonhole-Prinzip ist für Existenzbeweise auch in der Kombinatorik wichtig.
Triviale Probleme sind:

- In einer Menge von 13 Personen sind minestens zwei, die im selben Monat geboren sind.
- In der Schweiz (6'000'000 Bewohner) gibt es mindestens zwei Personen, welche dieselbe Anzahl Haar haben.
- Aus einer Menge von 10 blauen und 10 roten Socken müssen höchstens drei Socken blindlings ausgewählt werden, um ein Paar der gleichen Farbe zu haben.

Ein weniger triviales Beispiel ist folgendes: Zwei Freunde spielen folgendes Spiel: A wählt 10 Zahlen von 1 bis 40. B gewinnt, wenn er zwei verschiedene Untermengen der Kardinalität 3 in diesen 10 Zahlen findet, welche dieselbe Summe besitzen. Man beweise, dass B das Spiel immer gewinnen kann. Die Tauben seien alle Untermengen der Kardinalität 3. Davon gibt es also Binom(10,3)=120. Die Häuschen seien deren Summe. Die kleinste Summe ist 1+2+3=6, die grösste ist 38+39+40=117. Es gibt also 117-5=112 Summen (Häuschen). Da es mehr Tauben (Dreier-Mengen) als Häuschen (Summen) gibt, müssen mindestens zwei Tauben ins gleiche Häuschen (zwei Dreier-Mengen haben dieselbe Summe).

Das Diagonalisierungsprinzip

Theorem 3: Sei R eine binäre Relation in die Menge A. Die Diagonalmenge D von R ist gegeben durch $D = \{a \in A | (a,a) \notin R\}$. Sei für jedes $a \in A$ die Menge $R_a = \{b \in A | (a,a) \notin R\}$. Dann sind D und R_a für jedes a disjunkt.

Dieses Prinzip kann dazu verwendet werden, um zu zeigen, dass gewisse Mengen nicht abzählbar sind.

Beispiel: Die Potenzmenge 2^N der ganzen Zahlen ist nicht abzählbar.
Beweis: Angenommen sie wäre abzählbar, dann existiert eine Injektion f:N∅2^N

und die Potenzmenge könnte enumeriert werden als

$2^N=\{S_0, S_1, S_2,...\}$, sodass $S_i = f(i)$ mit $i \in N$.

Man betrachte nun die Menge

$D=\{n \in N | n \notin S_n\}$.

D ist eine Menge natürlicher Zahlen, und daher muss es ein $k \in N$ und ein S_k geben, welche der Menge D entspricht. Nun gilt: k ist in S_k oder k ist nicht in S_k. Angenommen

1) k ist in S_k ($k \in S_k$): Nach der Definition von D folgt, dass $k \notin D$, da aber $D = S_k$ ist, folgt, dass $k \notin S_k$. Ein Widerspruch.

2) $k \notin S_k$: Dann gilt nach der Definition von D, dass $k \in D$, somit folgt $k \in S_k$. Ein weiterer Widerspruch.

Da k weder in D noch nicht in D sein kann, muss geschlossen werden, dass es eine solche abzählbare Menge D nicht gibt. Also ist 2^N nicht abzählbar.

Das Diagolalisierungsprinzip ist fundamental in der Berechnungstheorie. Der obige Beweis ist typisch, um eine Funktion als nicht berechenbar zu beweisen.

3. ALPHABET UND FORMALE SPRACHEN

Ein **Alphabet** $\sum$ besteht aus endlichen **Symbolen** $\{a,b,c,d,...\}$. Über das Alphabet bildet man eine Halbgruppe (H,•) bezüglich der Relation •, welche die Verkettung (Konkatenation von Symbolen) bezeichne. Ein Element aus der Halbgruppe heisst **Wort**. Ist w ein Wort (oder Zeichenkette), so bezeichnet $|w|$ die Länge des Wortes (in Anzahl Symbolen). Die Menge aller Wörter wird mit $\sum^*$ bezeichnet. Das leere Wort e gehört auch zu $\sum^*$. Eine **Sprache** L ist eine Untermenge von $\sum^*$. Ist L eine Sprache, so ist L* die **Kleene'sche Hülle**. L* ist die Sprache, die man erhält, wenn in L alle Wörter null oder mehrere Male verkettet werden. Wenn v und w zwei Wörter sind, so wird die Konkatenation als vw geschrieben. Die Vereinigungsmenge der beiden wird abgekürzt als $v+w$ und bedeutet 'Wort v oder w' Ein Wort das n mal wiederholt wird, wird mit w^n abgekürzt (Beispiel: www ist w^3). w^* ist die Menge aller Wörter bestehend aus w, also $w^* = \{ e, w, ww, www, ...\}$. w^+ ist eine Abkürzung für ww^*. Wenn w ein Wort ist, so ist mit w^R das inverse Wort (w gelesen von rechts nach links) gemeint.

Von zentraler Bedeutung für die Berechnungstheorie ist die Repräsentation von Sprachen in einer finiten Form. Wie jede Menge kann auch eine Sprache durch eine Syntax wie $L(\{a,b\})=\{x\&x^*aa\}$ dargestellt werden. Diese Sprache L, beispielsweise, definiert die Sprache, die aus allen Wörtern des Alphabetes der Symbole a und b besteht, welche auf aa enden. Eine interessante Sprachfamilie sind die **regulären Sprachen**, die definiert sind durch die Menge der regulären

Ausdrücke als:

1. Ø und jedes Element aus ∑ ist ein regulärer Ausdruck
2. wenn *v* und *w* reguläre Ausdrücke sind, dann ist es auch (*vw*)
3. wenn *v* und *w* reguläre Ausdrücke sind, dann auch (*v*+*w*)
4. wenn *v* ein regulärer Ausdruck ist, dann auch *v**
5. nichts sonst ist ein regulärer Ausdruck

Die Sprache $L=\{a^*\}$ [abgekürzt als $L(a^*)$] besteht aus allen Wörtern bestehend aus dem Symbol *a*. Es ist somit $L(a^*) = \{e, a, aa, aaa, aaaa, ...\}$.

4. GÖDELISIERUNG

Wir haben gesehen, dass eine Sprache definiert wird als eine Menge. Ist diese Menge unendlich, so ist sie abzählbar, da wir voraussetzen, dass das Alphabet endlich ist. Daher kann eine injektive Abbildung (Injektion) zwischen den Wörtern einer Sprache und der Menge der natürlichen Zahlen konstruiert werden; d.h. die Wörter einer Sprache können durchnumeriert werden. Die Gödelisierung ist nichts anderes als eine solche Abbildung: Jedes Wort der Sprache wird in eine natürliche Zahl umgewandelt (kodiert). Die Entsprechung muss so sein, dass ein Algorithmus aus dem Wort die Zahl berechnen und aus der Zahl wiederum das Wort rekonstruieren kann.
Gegeben sei ein beliebiges, (endliches) Alphabet ∑. Eine injektive Funktion *h*: S*∅N heisst Göderisierung der Wörter *v* und *w* aus ∑*, wenn gilt

a) $v \neq w \Rightarrow h(v) \neq h(w)$ mit $v,w \in \sum^*$
b) *h(v)* ist in endlich vielen Schritten berechenbar
c) Für alle $n \in N$ ist es in endlich vielen Schritten feststellbar, ob es ein $w \in \sum^*$ gibt, sodass $h(w)=n$ ist.
d) Ist $n=h(w)$, so ist *w* in endlich vielen Schritten auffindbar. *h(w)* heisst die Gödelnummer des Wortes *w*.

Häufig verwendete Kodierungen sind die binäre, adische und die Primzahlenverschüsselung (siehe Stetter S.14ff). Allerdings ist die Art der Kodierung für die mathematische Fragestellungen der Berechenbarkeit unwichtig. In der Beweisskizze des Gödelschen Unvollständigkeitssatzes in diesem Artikel wird eine Primzahlenverschüsselung verwendet werden.

3. AUTOMATEN

Automaten sind Maschinenmodelle, welche von den physikalischen Eigenschaften von realen Maschinen abstrahieren. Automaten sind also in diesem Zusammenhang nur mathematische Hilfskonstruktionen, welche mit

den physikalischen Maschinen wenig zu tun haben. Nur die 'logische' Seite interessiert uns, aber nicht die konkrete, physikalische Konstruktion der Maschine. Die zentrale Frage dabei ist, welche (minimalen) Eigenschaften muss ein (abstrakter) Automat haben, damit er alle physikalischen Maschinen 'imitieren' kann? Gibt es überhaupt ein solches universelles Modell, sodass alle existierenden Maschinen auf dieses zurückgeführt werden können? Wenn es nämlich ein solches Modell gibt, so brauchen wir nur seine Eigenschaften zu studieren, um die Frage nach der maschinellen Berechenbarkeit zu beantworten.
Ein solches Automatenmodell kann in der Tat konstruiert werden. Man kann sich einen Automaten als eine Maschine vorstellen, die aus verschiedenen Magnetbändern (Eingabe-, Ausgabe- und Zwischenspeicher-band) und einer Verarbeitungseinheit, die in verschiedenen Zuständen sein kann. Wie schon gesagt, dass Magnetbänder von der heutigen technologischen Warte aus als veraltet angesehen werden können, spielt hier keine Rolle. Die physikalische Seite eines solchen Automaten ist unwichtig für die Fragen der Berechenbarkeit. Es geht uns hier nur um ein visuelles Bild von einem solchen Automaten.
Der hier noch nicht näher spezifizierte Automat arbeitet auf folgende Weise: Die Bänder sind der Länge nach in Felder aufgeteilt. Auf jedem Feld kann sich ein Symbol eines Alphabets befinden. Der Automat startet eine Verarbeitung, indem die Symbole auf den Bändern sequentiell gelesen/geschrieben werden und der Automat nach einer genau vorgegebenen Vorschrift nach jeder Lese-/Schreibaktion in einen bestimmten Zustand übergeht. Der Automat hält, wenn ein Haltezustand erreicht ist, sonst hält der Automat nicht. Es kann also sein, dass ein Automat in einer unendlichen Schaufe hängen bleibt.

Generell unterscheidet man **Automaten ohne Ausgabe** und **Automaten mit Ausgabe**. Automaten ohne Ausgabe sind Akzeptoren, d.h. solche die auf dem Eingabeband testen, ob ein bestimmtes Wort zu einer Sprache gehört oder nicht. Sie sind also dazu geeignet ein Ja-Nein-Problem zu lösen. Automaten mit Ausgabe sind Generatoren, d.h. sie beschreiben bei der Verarbeitung noch zusätzlich ein Ausgabeband. Es besteht kein prinzipieller Unterschied zwischen Automaten ohne und Automaten mit Ausgabe. Man kann sich leicht davon überzeugen, dass beide Typen vom Standpunkt der Berechenbarkeit dasselbe leisten.
Ferner werden **deterministische** und **nicht-deterministische** Automaten unterschieden. Deterministische Automaten sind solche, welche bei jedem

Zustandswechsel nur in einen einzigen Zustand übergeben können. Nicht-deterministische Maschinen können aus einer endlichen Menge von Zuständen nicht-deterministisch 'auswählen', in welchen Zustand sie übergeben. Der Verarbeitungspfad ist also nicht von vornherein deterministisch vorgegeben. Wie die 'Auswahl' gemacht wird, spielt keine Rolle. Nicht-deterministische Maschinen sind vom Standpunkt der Berechenbarkeit im allgemeinen gleichwertig wie deterministische. Dies ist intuitiv einzusehen, wenn eine nicht-deterministische Maschine durch die deterministische Maschine, welche alle nicht-deterministischen Zustandsübergänge sequentiell abarbeitet, simuliert wird. Aus Sicht der Komplexität hingegen scheint es einen Unterschied zu sein, welches Maschinenmodell verwendet wird. (Es ist immer noch offen, ob N=NP).

Vom Standpunkt der Berechenbarkeit unterscheiden man die vier Maschinenmodelle (**endlicher Automat (EA)**, **Kellerautomat (KA)**, **linear beschränkter Automat (LBA)** und die **Turingmaschine (TM)**), die den vier Sprachklassen von Chomsky entsprechen. Die TM ist die mächtigste Maschine in dem Sinne, dass sie alle andern simulieren kann, aber nicht umgekehrt. Die EA ist die schwächste Maschine, da sie von allen andern simuliert werden kann, aber selber keine andere simulieren kann. LBA kann eine KA simulieren, aber nicht umgekehrt. Wir haben also die Beziehung, wenn '>' heisst 'kann simulieren aber nicht umgekehrt':

$$TM > LBA > KA > EA$$

Die vier Automatenklassen sollen nun kurz vorgestellt werden. Weitere Modelle sind in der Literatur vorgeschlagen worden, die hier aber nur erwähnt werden sollen: die Post-Maschine und die 2-Kellerautomaten (siehe dazu Manna S.24ff). Beide Modelle sind aber mit der Turing-Maschine identisch, was ihre Berechenbarkeit betrifft, da sie sich gegenseitig simulieren können (siehe dazu Manne, Kapitel 1-2). Die Turing-Maschine ist, wie schon gesagt, die 'mächtigste' Maschine, in dem Sinne, dass alle andern durch diese simuliert werden können und *es war trotz vielen Versuchen nicht möglich einen Automaten anzugeben, welche durch die Turing-Maschine nicht simuliert werden könnte* (Church-Turing These). Insbesondere können alle heutigen Computer bis hinauf zu den komplexen, modernen Architekturen durch eine Turing-Maschine simuliert werden. Dies ist erstaunlich, wenn wir die Einfachheit einer Turing-Maschine betrachten. Es ist umso erstaunlicher, weil

Turing dieses Maschinenmodell konstuiert hat noch bevor es den ersten Computer gab (1936). Die Turing-Maschine ist daher zum Inbegriff der Berechenbarkeit geworden. “Berechenbar” heisst also in unserem Zusammenhang immer “auf einer Turing-Maschine-berechenbar”. “Ein Problem ist entscheidbar” heisst also immer “es gibt eine Turing-Maschine für unser Problem”. Es ist allerdings heute immer noch (oder wiederum) eine kontroverse Frage, ob der Begriff “auf einer Turing-Maschine-berechenbar” dasselbe sei wie “vom menschlichen Gehirn berechenbar”. Die Frage ist keineswegs trivial, wie Penrose (1989) in seinem faszinierenden Buch über - unter anderem - Computer- und menschliche Intelligenz zeigt. Aber lassen wir hier die philosophischen Implikationen und kehren zurück zu 'technischeren' Dingen.

1. ENDLICHER AUTOMAT (EA)

Ein endlicher (deterministischer) Automat ohne Ausgabe (EA) (finite automaton) wird definiert als ein Quintupel $(Q,\sum,\delta,q_0,F)$, wobei Q eine endliche Menge von **Zuständen**, $\sum$ ein endliches **Eingabealphabet**, δ die **Übergangsfunktion** δ: $Q\infty\sum\varnothing Q$, $q_0 \square Q$ der **Startzustand** und $F\prod Q$ die Menge der akzeptierenden **Endzustände** sind. Ein EA kann man sich vorstellen als eine Maschine bestehend aus einem (unbeschränkten) Eingabeband, das nur in eine Richtung gelesen werden kann und einem Eingabekopf, der folgende Schritte wiederholt ausführt: lesen eines Zeichens (Symbols) auf dem Band, ein Zeichen nach rechts gehen und in einen andern Zustand übergehen.

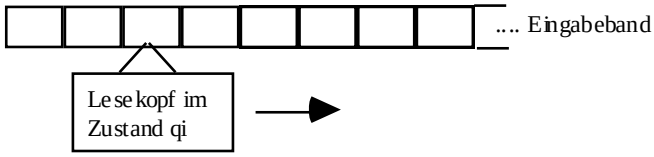

Abbildung 1

Der Ausgangspunkt einer Abarbeitung ist, dass der EA sich im Zustand q_0 befindet und ein Wort auf dem Band steht. Der Rest des Bandes ist mit einem Blank - hier mit # bezeichnet - aufgefüllt. Der Kopf zeigt auf das erste Zeichen des Wortes. Man sagt, dass der EA das Wort *akzeptiert*, wenn der Lesekopf am rechten Ende des Wortes auf einem # steht und sich in einem akzeptierenden Zustand $q_f \square F$ befindet. Ein EA, der genau die Wörter einer Sprache L akzeptiert, akzeptiert die Sprache L.

Beispiel (Herschel S.24): Es soll ein EA konstruiert werden, welcher die Sprache L({a,b}) akzeptiert, die nur aus dem Wort *b* besteht. Wir haben

$\sum$={a,b}. Es sei ferner Q={q_0, q_1, q_2}, F={q_1} und die Übergangsfunktion sei definiert als:

$\delta(q_0,a)=q_2$ $\delta(q_0,b)=q_1$
$\delta(q_1,a)=q_2$ $\delta(q_1,b)=q_2$
$\delta(q_2,a)=q_2$ $\delta(q_2,b)=q_2$

Es ist leicht einzusehen, dass diese EA genau die Sprache L akzeptiert: Die EA befindet sich am Anfang im Zustand q_0. Wenn das erste Zeichen ein *a* ist, so geht der EA in den Zustand q_2 über. In diesem Zustand bleibt er, unabhängig davon, welche weiteren Zeichen gelesen werden. Ist das erste Zeichen hingegen ein *b*, so geht der EA in den Zustand q_1 über. Folgen weitere Zeichen, so geht der EA wiederum in den Zustand q_2 über. Da der EA also nur im Endzustand q_1 hält, wenn genau ein *b* gelesen wurde, so akzepiert der Automat genau das Wort *b*, denn nur q_1 ist ein akzeptierender Zustand, q_2 hingegen ist kein akzeptierender Zustand.
Ein EA kann auch als Übergangsgraph beschrieben werden: Die Knoten sind die Zustände und die Kantenmenge ist die Übergangsfunktion. Die akzeptierenden Endzustände werden durch einen dicken Kreis dargestellt. Unser Automat L kann somit als Diagramm (Abbildung 2) dargestellt werden.

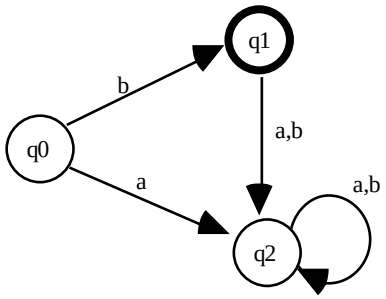

Abbildung 2

Wird ein Wort von einem EA verarbeitet, so brauchen wir nur den Kanten zu folgen. Wenn der Automat in Abbildung 2 das Wort *aabb* verarbeitet, so folgen wir ausgehend von q_0 der Kante, die mit *a* beschriftet ist und gelangen zu q_2. Dann folgen wir dreimal der Kante *a,b* die uns jeweils zu q_2 zurückführt. Wir bleiben also in q_2. Die akzeptierende Zustandsmenge F besteht aber nur aus q_1. Da wir uns also nach der Verarbeitung von *aabb* nicht in einem akzeptierenden Zustand befinden, so wird das Wort *aabb* von unserem Automaten nicht akzeptiert.

Moore-Maschinen sind EAs *mit* Ausgabe. Ein Beispiel ist das Programm, welches zu einer binären Zeichenkette den Rest modulo drei berechnet (siehe Hopcroft S.45). Wir gehen hier nicht näher darauf ein.

Zwei Automaten sind **äquivalent**, wenn sie dieselbe Sprache akzeptieren. Eine interessante Frage in Bezug eines gegebenen EA ist, welches der dazu minimale, äquivalente EA ist, d.h. eine EA mit einer minimalen Anzahl Zuständen.
Eine andere Frage ist, welche **Sprachklassen** ein EA akzeptieren kann. Man kann z.B. leicht einsehen, dass es keinen endlichen Automaten geben kann, welcher die Sprache S({a,b}) = {w| w=$a^n b^n$, mit n□N} akzeptiert. Denn angenommen ein solcher EA existierte, dann kann ein Diagramm mit q (=|Q|) Zuständen erstellt werden. Da aber ein EA mit n>q gleichen Zeichen nach maximal q Übergängen - sagen wir nach exakt p≤q Übergängen - wieder in denselben Zustand zurückkehren muss, kann er eine Eingabe $a^n b^n$ von einer Eingabe $a^{n+p} b^n$ nicht unterscheiden. Ein EA, welcher die Eingabe $a^n b^n$ akzeptiert, müsste also auch die Eingabe $a^{n+p} b^n$ akzeptieren. Daher kann die Sprache {w| w=$a^n b^n$, mit n□N} von keinem EA erkannt werden.

Ein **nicht-deterministischer** endlicher Automat (NEA) unterscheidet sich von einem EA durch die Übergangsfunktion. Diese ist für einen NEA definiert als $\delta: Q \infty \sum \varnothing 2^Q$. Für jeden NEA kann ein EA konstruiert werden (der Beweis findet sich in Hopcroft p.23). Dies bedeutet, dass die Sprachklasse, welche von einem NEA erkannt wird, auch von einem EA erkannt wird. NEA und EA sind daher vom Standpunkt der Berechenbarkeit äquivalente Automaten. Selbst nicht-deterministische einfache Automaten mit e-Übergängen (eNEA) sind zur Klasse der EAs äquivalent. eNEA sind solche NEA, welche in einen andern Zustand übergehen können, ohne dass ein Zeichen vom Eingabeband gelesen wird.

2. KELLERAUTOMAT (KA)

Wir haben gesehen, dass die Sprache S({a,b}) = {w| w=$a^n b^n$, mit n□N} von keinem EA akzeptiert werden kann. Der Grund liegt offenbar darin, dass der EA keinen Zwischenspeicher besitzt, wo die Anzahl der gelesenen Symbole *a* der Sprache S memorisiert werden könnte. Der EA hat kein Gedächtnis! Dies kann auch festgestellt werden, wenn man versucht, ein Zustandsdiagramm für den EA, der die Sprache S akzeptieren soll, zu konstruieren. Die Abbildung 3 zeigt, dass dies nicht gelingen kann. Diese Abbildung gibt zwar das Diagramm richtig wieder für n≤3, aber nicht für ein beliebiges n. Man sieht leicht ein, dass

das Diagramm in seine horizontale und vertikale Dimension unbeschränkt erweitert werden müsste, um das Zustanddiagramm zu konstruieren.

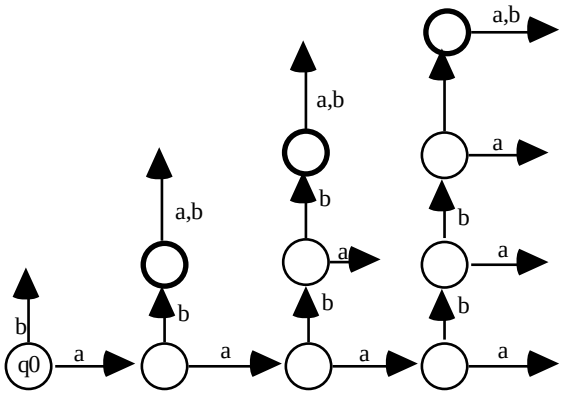

Abbildung 3 (aus Herschel S.27)

Um eine 'mächtigere' Maschine zu konstruieren, fügen wir dem EA einen Kellerspeicher hinzu. Der Kellerspeicher ist ein Lese-/Schreibeband, das in der LIFO (last-in-first-out) Manier beschrieben wird: Wird eine Zeichen auf das Kellerband geschrieben, so wird sein Lese-/Schreibkopf hernach um eine Position in Richtung 'Top' verschoben.

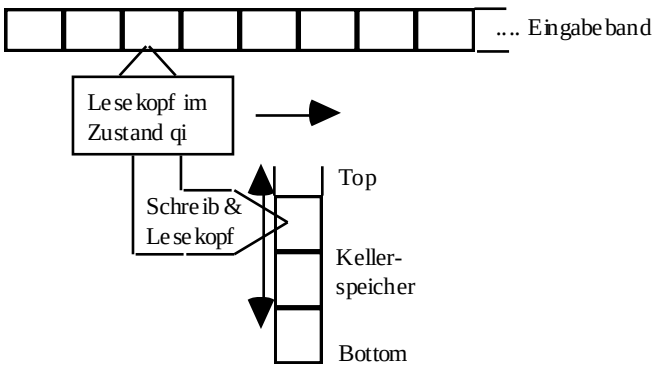

Abbildung 4

Wird eine Zeichen vom Kellerband gelesen, so wird zuerst eine Position Richtung 'Bottom' gefahren und dann gelesen. Anfangs ist der Kopf auf die Position 'Bottom' gesetzt (siehe Abbildung 4).
Formell kann dieser Automat, der Kellerautomat (KA) (pushdown automaton) heisst, als 7-Tupel $(Q,\sum,\Gamma,\delta,q_0,z_0,F)$ definiert werden, wobei Q, $\sum$, q_0 und F wie beim EA zu interpretieren sind. Γ ist das Kelleralphabet, $z_0 \Box \Gamma$ ist das Anfangssymbol, δ ist die Übergangsfunktion δ: $Q\infty(\sum\approx\{e\})\infty\Gamma\varnothing 2^{Q\infty\Gamma^*}$.
Eine von einem KA akzeptierten Sprache kann mit zwei Methoden definiert

werden:
1) sie ist die Menge der Eingaben (=Wort auf den Eingabeband), welche den Kellerinhalt leert.
2) sie ist - ähnlich wie beim EA - die Menge der Eingaben, bei denen der Automat in einem akzeptierenden Endzustand hält.
Die beiden Definitionen sind äquivalent (Hopcroft S.117).

Es kann nun ein Kellerautomat (KA) konstuiert werden, welcher die Sprache $S(\{a,b\}) = \{w \subseteq w=a^n b^n$, mit $n \square N\}$ akzeptiert. Dieser hat folgende Definition (Herschel S. 57):
$\sum=\{a,b\}$, $Q=\{q_0,q_1,q_2\}$, $\Gamma=\{z_0,a,e\}$, $F=\{q_2\}$ und die Übergangsfunktion

$\delta(q_0,a,k)=\{(q_0,ka)\}$ $\quad$ $\delta(q_0,a,a)=\{(q_0,aa)\}$
$\delta(q_0,b,a)=\{(q_1,e)\}$ $\quad$ $\delta(q_1,b,a)=\{(q_1,e)\}$
$\delta(q_1,e,k)=\{(q_2,e)\}$

Die Übergangsfunktion $\delta(q_0,a,k)=\{(q_1,ka)\}$ z.B. kann interpretiert werden wie folgt: ein KA im Zustand q_0 und dem Eingabesymbol a unter dem Lesekopf mit den Kellersymbol k unter dem Lese/Schreibkopf geht in den Zustand q_1 über, wobei das Kellersymbol durch ka ersetzt wird.
Man beachte dass der KA als nicht-deterministische Maschine definiert ist. Im Gegensatz zum EA, der mit der NEA identisch ist, gilt dies für den Kellerautomat nicht: der deterministische Kellerautomat (DKA), der sich nur durch die Übergangsfunktion $\delta: Q\infty(\sum\approx\{e\})\infty\Gamma\varnothing Q\infty\Gamma^*$ vom KA unterscheidet, ist weniger mächtig. Dies kann man sehen, wenn die beiden Sprachen $S_1(\{a,b,c\})=\{wcw^R \subseteq w \square \{(a+b)^*\}$ und $S_2(\{a,b\})=\{ww^R \subseteq w \square \{(a+b)^*\}$ verglichen werden (Hopcroft S.118). Ein DKA kann die Sprache S_1 zwar akzeptieren, da deterministisch der Zeitpunkt feststeht, wann der Keller wieder reduziert werden soll, nämlich dann, wenn ein c gelesen wird. Für die Sprache S_2 muss ein nicht-deterministischer Kellerautomat (KA) verwendet werden, da deterministisch nicht entschieden werden kann, wann der Keller wieder abzubauen ist.

Kellerautomaten sind Maschinen, die eine hierachische Struktur erkennen können. Zum Beispiel kann ein KA konstruiert werden, welche die richtige Klammerung von Ausdrücken erkennt. Er ist definiert als: $\sum=\{[,]\}$, $Q=\{q_0,q_1\}$, $\Gamma=\{z_0,[,e\}$, $F=\{q_1\}$ und der Übergangsfunktion

$\delta(q_0,[,k)=\{(q_0,k[)\}$ $\delta(q_0,[,[)=\{(q_0,[[)\}$
$\delta(q_0,],[)=\{(q_0,e)\}$ $\delta(q_0,e,k)=\{(q_1,e)\}$

(siehe Herschel Aufgabe 1.2/3 S. 66).

Auch bei Kellerautomaten kann wiederum die Frage gestellt werden, welcher Automat eine bestimmte Sprache mit minimalen Zuständen akzeptiert. Hier gilt allerdings, dass die Verkleinerung der Anzahl Zustände im allgemeinen durch eine Vergrösserung des Kelleralphabets kompensiert werden muss. Es gilt sogar der Satz: Ist KA ein Kellerautomat mit n Zuständen und m Kellerzeichen, so gibt es einen Kellerautomaten KA' mit nur einem Zustand und n^2m+1 Kellerzeichen, sodass KA und KA' beide dieselbe Sprache akzeptieren (Satz 1.2/2 in Herschel S. 65).
Eine noch wichtigere Frage ist, ob es Sprachen gibt, welche durch keinen KA akzeptiert werden. Dies gibt es in der Tat. Ein solche ist die folgende Sprache L:

$L(\{a,b,c\}) = \{w \subseteq w=a^nb^nc^n, \text{ mit } n \square N\}$

Oben wurde ein KA für die Sprache $S=\{w \subseteq w=a^nb^n, \text{ mit } n \square N\}$ angegeben. Die Lösung bestand darin, dass zuerst die Symbole a^n abgekellert werden, und die Symbole b^n leeren den Keller wieder. Dabei wird der Keller bei gleicher Anzahl Symbole a und b exakt geleert. Dieselbe Lösung kann aber für die Sprache L nicht mehr verwendet werden, weil nichts im Keller übrig bleibt, um die Anzahl Symbole c in c^n zu testen. Werden aber die Symbole a^nb^n gekellert, so kann die gleiche Zahl zwischen den Symbolen a und b nicht getestet werden. Wie wir auch kellern, die Sprache L kann offenbar von keinem KA akzeptiert werden. Dies kann mit Hilfe des Pumping Lemmas leicht bewiesen werden (zum Beweis siehe Hopcroft S. 135).
Es wäre naheliegend, für die Lösung des Problems einen Automaten vorzuschlagen, der 2 Kellerbänder enthält. Man könnte dann die Symbole a im ersten und die Symbole b im zweiten Keller abkellern und beim Lesen der Symbole c beide Keller simultan entkellern. Man kann leicht zeigen, dass ein solcher Automat, welcher 2-Kellerautomat (2KA) heisst, die Sprache L akzeptieren kann. Wir wollen diese Lösung jedoch nicht weiter verfolgen, da ein 2KA äquivalent zu einer Turing-Maschine ist, d.h. jeder 2KA kann durch eine Turing-Maschine simuliert werden *und umgekehrt*. Da Turing-Maschinen (TM) aber das überzeugendere Maschinenmodell sind, sollen diese im nächsten Abschnitt behandelt werden.

3. TURINGMASCHINEN (TM)

Eine Turingmaschine (TM) besitzt ein einseitig unbeschränktes Eingabeband, das gleichzeitig auch beschrieben werden kann, und einen Lese/Schreibkopf, welcher sich in beide Richtungen bewegen kann (Abbildung 5). Auf den ersten Blick scheint also eine TM dem EA sehr ähnlich zu sein. Man beachte jedoch drei wichtige Unterschiede: a) Das Eingabeband ist gleichzeitig Schreibband, b) der Lesekopf ist gleichzeitig auch Schreibkopf, und c) er kann in beide Richtungen fahren. Beim EA konnte der Lesekopf nur nach rechts fahren.

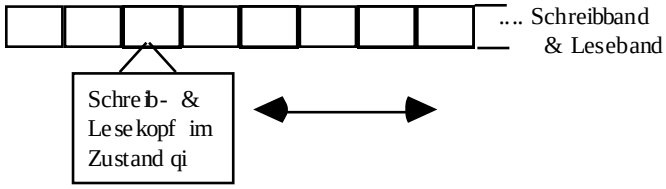

Abbildung 5

Formal ist eine TM definiert als $M=(Q,\sum,\Gamma,\delta,q_0,\#,F)$, wobei Q, $\sum$, q_0 und F definiert sind wie beim EA. Γ ist die Menge der erlaubten Bandsymbole, $\#\square\Gamma$ ist das Blanksymbol auf dem Band und δ ist die Übergangsfunktion $\delta: Q\infty\Gamma\varnothing Q\infty\Gamma\infty\{L,R,H\}$. Ein Übergang $\delta(q_1,a)=(q_2,b,R)$ kann interpretieret werden wie folgt: Eine TM im Zustand q_1 und dem Bandsymbol a unter seinem Kopf, überschreibt dieses Symbol durch b, geht in den Zustand q_2 über und verschiebt seinen Kopf um eine Position nach rechts (R=rechts, L=Links, H=bleibt stehen). Die Übergangsfunktion kann auch als eine Matrix - genannt Maschinentafel - angegeben werden, in der auf der Horizontalen das Bandsymbol unter dem Kopf und auf der Vertikalen der Zustand abgetragen ist.

Beispiel: Gesucht eine TM, deren Kopf auf einer 1 steht, soll rechts das erste 0 auf dem Band suchen und dieses mit einer 1 überschreiben und dann stehen bleiben. Die Maschine ist definiert als: $\sum=\{0,1\}$, $Q=\{q_0,q_1\}$, $F=\{q_1\}$, mit der Maschinentafel

	0	1
q_0	$1q_1H$	$1q_0R$
q_1	$0q_1H$	$1q_1H$

Die einzelnen Instruktionen, welche eine TM ausführen kann, sind recht primitiv. Es ist daher zweckmässig Elementarmaschinen zu konstruieren, die eine Anzahl von Instruktionen ausführen (z.B. zur ersten 0 rechts auf dem Band

gehen), und diese Maschinen dann zu koppeln. Die Lösungsbeschreibung von komplexeren Problemen kann dadurch stark vereinfacht werden. (Für eine ausführlichere Darstellung dazu siehe Herschel S. 71).

Man hat versucht, die Turingmaschine in verschiedene Weise zu einem 'mächtigeren' Automatenmodell auszubauen in dem Sinne, dass die verbesserte Maschine Sprachklassen erkennen könnte, welche die einfache TM nicht erkennen kann, wie wir vom EA zum KA dann zur TM übergegangen sind. Es wurde beispielsweise

a) das Band beidseitig unbeschränkt gemacht (bTM)
b) mehrere Lese/Schreibköpfe auf einem Band hinzugefügt (kTM)
c) mehrspurige Bänder mit mehreren Lese/Schreibköpfe eingeführt (mkTM)
d) zweidimensionale Bänder verwendet
e) nicht-deterministische Modelle (NTM) diskutiert

Die Erweiterungen können beliebig variert werden. So kann ein nicht-deterministisches Maschinenmodell mit beitseitig unbeschränktem Band mit mehreren Köpfen konstruiert werden (bkNTM). Die Erweiterungen waren jedoch immer ein Fehlschlag in dem Sinne, dass die dadurch verbesserten Turingmaschinen durch eine einfache TM simuliert werden können. Insbesondere ist die Klasse der von NTM akzeptierten Sprachen nicht grösser als die Klasse der TM akzeptierten Sprachen. (NTM spielen hingegen in der Komplexitätsanalyse eine wichtige Rolle). Wir wollen die Menge der Sprache, welche von einem TM akzeptiert wird, als **rekursiv aufzählbare Menge** bezeichnen. Es ist zweckmässig eine Sprachklasse - die **rekursive Menge** - zu definieren, die mindestens von einer TM erkannt werden, die auf alle Eingaben stoppt. Die rekursive Menge ist eine echte Untermenge der rekursiv aufzählbaren Menge.
Es ist also nicht gelungen ein 'mächtigeres' Maschinenmodell zu finden, welches die TM nicht simulieren könnte. Damit sind wir mit den Turingmaschinen an eine hierachische Spitze von (maschineller) Berechenbarkeit gelangt: Ein Problem, welches von einer TM nicht berechnet werden kann, kann auch von keiner andern Maschine (auch dem Gehirn?) nicht berechnet werden.

4. LINEAR BESCHRÄNKTER AUTOMAT (LBA)

Obwohl linear beschränkte Automaten (LBA) weniger leistungsfähig sind als Turingmaschinen, werden diese hier nach den Turingmaschinen behandelt,

ganz einfach deshalb, weil sich LBAs besser als Spezialfall von TMs betrachtet werden können. Ein LBA ist eine TM, welche auf dem Band rechts nicht über ein Spezialzeichen - wir nennen es $ - hinauslesen und -schreiben darf und links durch ein zweites Spezialzeichen ¢ beschränkt ist. Die Zeichen werden am Anfang der Abarbeitung auf das Band geschrieben (Abbildung 6).

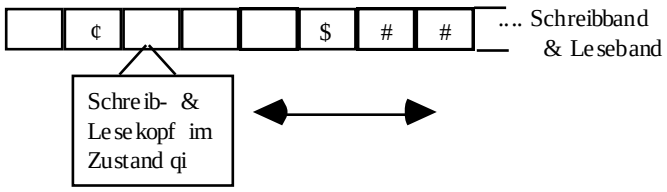

Abbildung 6: LBA

Hat das Eingabewort auf dem Band die Länge n, so ist die Länge k des beschreibaren Bandes - also die Anzahl der Felder zwischen ¢ und $ - eine lineare Funktion von n: k=a*n+b. Es gilt der Satz: Ist die Länge des Bandes eines Automaten LBA gleich k=a*n+b, so gibt es einen dazu äquivalenten Automaten LBA' mit a=k und b=0. D.h. das Band kann immer auf die Länge des Eingabewortes beschränkt werden. (Herschel S.82).
Für die Sprache L({a,b,c}) = {w⊆w=$a^n b^n c^n$, mit n□N}, die von keinem KA akzeptiert werden kann, kann man leicht einen (deterministischen) LBA konstruieren (siehe Herschel S. 83).
Die Frage, ob deterministische LBAs und nicht-deterministische LBAs dieselbe Sprachklasse akzeptieren, blieb lange Zeit offen. Ihre Gleichheit wurde erst in den 70iger Jahren bewiesen (siehe Herschel Satz 1.4/2 S. 83).

Die Sprachklasse, welche von einem LBA akzeptiert wird, ist eine echte Untermenge der rekursiv aufzählbaren Menge. Dies zu zeigen fällt nicht schwer und beruht auf der Diagonalisierung. Sie ist aber auch eine echte Unterklasse der rekursiven Mengen. Ein Beweis findet sich in Hopcroft S.247ff.

4. GRAMMATIKEN

Eine Grammatik ist ein Quadrupel $G=(V,T,P,S)$, wobei V und T endliche disjunkte Mengen von **Variablen** (Nicht-Terminale) und **Terminale** sind. P ist eine endliche Menge von **Produktionen**, welche die Form $\alpha \to \beta$ haben, wobei α und β Zeichenketten von Symbolen aus $(V \cup T)^*$ sind (mit α≠e). S ist eine spezielle Variable und heisst **Startsymbol**.
Beispiel 1: die Grammatik der arithmetischen Ausdrücken mit Addition und

Multiplikation kann durch die folgende Grammatik G_a ausgedrückt werden:

$$G_a = (\{E\}, \{+, *, (,), id\}, P, E)$$
$$\text{mit } P = \{E \to E + E, E \to E * E, E \to (E), E \to id\}$$

Eine Grammatik G definiert eine Sprache L(G) in dem Sinne, dass aus dem Startsymbol und den Produktionen alle zur Sprache L gehörenden Wörter (bestehend nur aus Terminalen) hergeleitet werden können. Zum Beispiel kann das Wort $(id + id) * id)$ aus der Grammatik G_a hergeleitet werden durch folgende Ableitung, wobei der Operator □ für diesen Zweck verwendet wird:

$$P \Rightarrow E * E \Rightarrow E * id \Rightarrow (E) * id \Rightarrow (E + E) * id \Rightarrow (E + id) * id \Rightarrow (id + id) * id$$

Bei jedem Übergang □ wird eine Produktion angewendet. Die Sprache $L(G_a)$ besteht also aus allen legalen arithmetischen Ausdrücken bezüglich der Addition und Multiplikation. Eine Grammatik ist also eine endliche Beschreibung einer meist unendlichen Menge (von Wörtern).
Es sollen im folgenden vier Grammatikklassen kurz vorgestellt werden, welche den Sprachklassen entsprechen, denen wir schon bei den Automaten begegnet sind.

1. REGULÄRE GRAMMATIK (CHOMSKY TYP 3)

Haben die Produktionen alle die Form $A \to wB$, $A \to w$, wobei A und B Variablen sind, und w ein Wort aus T^* ist, so nennt man die Grammatik rechts-linear. Wenn alle Produktionen die Form $A \to Bw$, $A \to w$ haben, so nennt man die Grammatik links-linear. Eine rechts- oder links-lineare Grammatik heisst auch **reguläre Grammatik**. Die Klasse der reguläre Grammatiken GR definiert eine Sprachklasse L(GR).
Satz: L(GR) ist mit der Klasse der **regulären Sprachen** identisch. Es gilt sogar ein noch wichtigerer Satz: Die L(GR) ist mit der Sprachfamilie, welche durch einen EA akzeptiert werden, identisch.
Für reguläre Sprachen gelten wichtige Gesetze: Sind S, S_1 und S_2 reguläre Sprachen, so sind auch die Sprachen $S_1 \leftrightarrow S_2$, $S_1 \approx S_2$, $\sum^* - S$, S^*, $S_1 S_2$ regulär. (S^* ist die Kleen'sche Hülle und $\sum^* - S$ ist das Komplement, während $S_1 S_2$ dadurch entsteht, dass die Wörter in S_1 und S_2 konkateniert werden).
Ein endlicher Automat kann also genau dann entscheiden, ob ein Wort zu einer Sprache gehört, wenn die Sprache durch eine reguläre Grammatik beschrieben werden kann. Reguläre Sprachen werden vor allem in der lexikalischen Analyse eines Compilers verwendet.

2. KONTEXTFREI GRAMMATIK (TYP 2)

Sind die Produktionen alle von der Form $A \to \alpha$, wobei A eine Variable und α ein Wort aus (V≈T)* ist, so wird die Grammatik **kontextfrei** genannt. Die Sprachklasse, welche durch kontextfreie Grammatiken beschrieben werden können, werden **kontextfreie Sprachen** (kfS) genannt. Es gilt: Die Sprachklasse, welche von einem KA erkannt (akzeptiert) werden, ist die Klasse der kontextfreien Sprachen (Beweis Hopcroft S. 121ff). Wenn $\alpha \neq e$, so kann jede kontextfreie Grammatik in die Chomsky-Normalform (CNF) überführt werden, für welche die Produktionen von der Form $A \varnothing BC$ oder $A \varnothing a$ sind, wenn A, B und C Variablen sind, und a ein Terminal ist (Hopcroft S.99). Die CNF spielt für verschiedene Beweise eine Rolle: Da das Problem, ob zwei kontextfreie Grammatiken dieselbe Sprache definieren, im allgemeinen unentscheidbar ist (siehe Tabelle 1, S.25), hat die Überführung der Grammatik in eine Normalform Vorteile, da die Grammatiken dann eher verglichen werden können.
Wie für reguläre Sprachen gelten auch für kontextfreie Sprachen wichtige Gesetze: Sind S, S_1 und S_2 kontextfreie Sprachen, so sind auch die Sprachen $S_1 \approx S_2$, S* und $S_1 S_2$ kontextfrei. Andererseits ist die Schnittmenge von zwei kontextfreien Sprachen, sowie das Komplement einer kontextfreien Sprache im allgemeinen keine kontextfreie Sprache mehr. Das Komplement einer *deterministischen*, kontextfreien Sprache dkfS ist jedoch wieder eine dkfS.
Interessant ist auch, dass es Algorithmen gibt, welche für eine gegebene kontextfreie Sprache entscheiden, ob diese leer, endlich oder unendlich ist (siehe Tabelle 1). Auch die Frage, ob ein beliebiges Wort aus $\sum^*$ zur einer kontextfreien Sprache S gehört oder nicht, ist entscheidbar. Ein berühmter Algorithmus dafür ist der Cocke-Younger-Kasami (CYK)-Algorithmus (Hopcroft S.149).

Deterministische, kontextfreie Sprachen (dkfS) (solche, die von einem DKA erkannt werden) spielen in der Kompilertheorie eine wichtige Rolle. Viele Programmiersprachen können mit den Mitteln einer dkfS beschrieben werden. Beschränkte Formen dazu sind die LR- oder die LL-Grammatiken. Für Compiler-Generatoren spielen diese Klasse eine wichtige Rolle. Interessant ist, dass das Komplement von einer dkfS wieder eine dkfS ist (Hopcroft S.260). Die Frage jedoch, ob zwei dkfS äquivalent sind, ist ein wichtiges ungelöstes Problem in der Sprachtheorie, in dem Sinne, dass man nicht weiss, ob das Problem entscheidbar ist oder nicht.

3. KONTEXT SENSITIVE GRAMMATIK (TYP 1)

Sind die Produktionen alle von der Form $\alpha \rightarrow \beta, \text{ mit } |\alpha| \leq |\beta|$, wobei α und β Zeichenketten aus (V≈T)* sind (mit α≠e), so wird die Grammatik **kontextsensitiv** genannt. Die Sprachklasse, welche durch kontextsensitive Grammatiken beschrieben werden können, werden **kontextsensitive Sprachen** (ksS) genannt. Es gilt der Satz: Die Sprachklasse, welche von einem LBA erkannt werden, ist der Klasse der kontextsensitiven Sprachen äquivalent.
Als Beispiel für eine kontextsensitiven Grammatik der Sprache L({a,b,c}) = {w⊆w=$a^n b^n c^n$, mit n□N} gibt Herschel die Produktionen an (S. 130).
Es ist immer noch offen, ob das Komplement einer ksS wieder eine ksS ist. Interessant ist aber, dass die Schnittmenge von beide ksS wieder eine ksS ist.
Jede kontextsensitive Sprache ist rekursiv und damit eine echte Untermenge der freien Grammatiken (siehe nächster Abschnitt). Allerdings gibt es auch rekursive Sprachen, die nicht kontextsensitiv sind (Herschel S. 134 und Hopcroft S. 248).

4. FREIE GRAMMATIK (TYP 0)

Gibt es keine Einschränkung der Produktionsformen, so sprechen wir von **freien Grammatiken**. Man nennt diese auch **Semi-Thue-Systeme**, **Typ-0-Grammatiken**, **allgemeine Regelgrammatiken** oder **nicht eingeschränkte Grammatiken**. Es gilt: Ist L(G) eine freie Grammatik, so ist L eine rekursiv aufzählbare Sprache und umgekehrt. (Beweis: Hopcroft S.241). Das bedeutet, dass es zu jeder Turing-Maschine T eine freie Grammatik G gibt, sodass L(T)=L(G). Freie Grammatiken sind die mächtigste Klasse von Grammatiken, die alle andern umfasst, da die Produktionen in keiner Weise eingeschränkt sind.

Zusammenfassend können wir eine Hierarchie von Sprachklassen über ein Alphabet ∑ (Mengenklassen) angeben (Abbildung 7). Wichtig dabei ist die Bemerkung, dass es auch nicht rekursiv-aufzählbare Sprachen gibt.

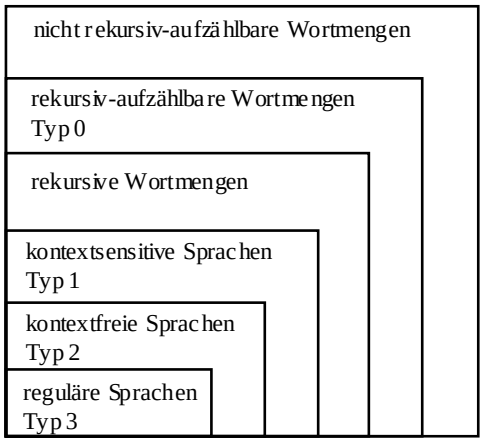

Abbildung 7: Hierachie der Teilmengen von ∑*

(Ein weiteres Schema, welches P und NP eingezieht siehe in Wagner S. 224). Einige Gesetze sind in Tabelle 1 zusammengestellt.

		Problem	x□L	L=Ø	L=∑*	$L_1=L_2$	$L_1 \leftrightarrow L_2$= Ø $L_1 \sqcap L_2$
Sprachklasse	Grammatik	Automat					
regulär	regulär	EA=NEA =eNEA	E	E	E	E	E
deterministisc h kontextfrei		DKA	E	E	E	??	U
kontextfrei (nicht-det.)	kontextfrei	KA(+NKA)	E	E	U	U	U
kontext-sensitive	kontext-sensitive	LBA	E	U	U	U	U
rekursiv			E	U	U	U	U
rekursiv aufzählbar	frei=Semi-Thue	TM=NTM =2-KA	U	U	U	U	U

(siehe Mandroli p.182, und Hopcroft S.306).
E=entscheidbar, U=unentscheidbar, ?? = noch nicht bekannt

Tabelle 1

Es ist bemerkenswert, dass mehr als die Hälfte aller in der Tabelle 1 angegebenen Probleme unentscheidbar sind. Wir werden später noch weitere solche Probleme sehen.

5. MENGEN UND FUNKTIONEN

Turing Maschinen können als *Akzeptoren* oder als Gereratoren betrachtet werden. Als Akzeptoren wird der Maschine ein String (Wort) als Eingabe auf das Band geschrieben. Die Maschine testet dann, ob das Wort zur Sprache, für welche die Maschine konstruiert ist, gehört oder nicht. Gehört das Wort zur Sprache, so hält die Maschine in einem akzeptierenden Zustand, sonst hält sie in einem nicht-akzeptierenden Zustand oder hält nie an. Als *Generatoren* werden der Maschine eine Liste von Worten (getrennt durch ein Spezialbandzeichen), die den Parametern einer Funktion f entsprechen, als Eingabe auf das Band geschrieben. Die Maschine berechnet den Funktionswert, der auf das Band zurückgeschrieben wird, zu diesen Argumenten, falls der Wert existiert. Im ersten Falle ist also die Turing-Maschine geeignet die Zugehörigkeit eines Elements zu einer *Menge* zu bestimmen; im zweiten Falle hingegen manipuliert die Turing-Maschine eine *Funktion*. Es ist nun interessant zu erforschen, welche Mengen- und Funktionsklassen eine Turing-Maschine manipulieren kann.

REKURSIVE UND REKURSIV AUFZÄHLBARE MENGEN

Wir haben eine Menge als **rekursiv aufzählbar** *definiert*, wenn diese von einer Turing-Maschine akzeptiert wird. Genauer: Gegeben sei eine Sprache L. Diese ist genau dann rekursiv aufzählbar, wenn es eine Turing-Maschine gibt, die auf alle Wörter als Eingabe in einem akzeptierenden Zustand hält, genau dann wenn das Wort zur Sprache gehört, andernfalls (wenn das Wort also nicht zur Sprache gehört) hält die Maschine in einen nicht-akzeptierenden Zustand oder sie hält nie. Eine Menge wurde als **rekursive** Menge *definiert*, wenn es eine Turing-Maschine gibt, die bei *allen* Eingaben stoppt; und zwar in einem akzeptierenden Zustand, wenn das Eingabewort auf dem Band zur Menge gehört, oder in einem nicht-akzeptierenden Zustand, wenn das Eingabewort nicht zur Menge gehört.

Betrachten wir diese Mengen etwas genauer. Man kann zunächst festhalten, dass die Menge aller Wörter, die zu einer (unendlichen) Sprache L mit einem endlichen Alphabet $\sum$ gehören, abzählbar sind. Wir brauchen die Wörter in aufsteigender Symbollänge nur durchzunumerieren, um eine Entsprechung zwischen den Anzahl Wörtern und der Menge der natürlichen Zahlen zu haben. Voraussetzung einer solchen Entsprechung ist ein Algorithmus, der aus dem Wort seine Zahl berechnet und umbekehrt (Gödelisierung). Rekursiv aufzählbare sowie rekursive Mengen können daher ganz unabhängig von Wortmengen untersucht werden, die Menge der natürlichen Zahlen genügt.

Einfache Beispiele von rekursiv aufzählbaren Mengen sind z.B.

Die Menge der geraden Zahlen : { 0, 2, 4, 6, 8, ... }

Die Menge der Quadratzahlen : { 0, 1, 4, 9, 16, ... }

Die Menge der Primzahlen : { 2, 3, 5, 7, 11, ... }

da diese Mengen offenbar mit einer Turing-Maschine (bzw. einem Algorithmus) erzeugt werden können. Für jede der drei Mengen kann auch das Komplement (die Menge der Elemente, die nicht zur Menge gehören) durch eine Turing-Maschine erzeugt werden. Das heisst auch das Komplement der drei Mengen ist rekursiv aufzählbar. Tatsächlich kann für jede Zahl entschieden werden, ob sie gerade oder ungerade, eine Primzahl oder keine Primzahl, eine Quadratzahl oder keine Quadratzahl ist. Rekursiv aufzählbare Mengen, deren Komplement auch rekursiv aufzählbar sind, heissen **rekursiv**. Dies stimmt mit der obigen Definition überein, da für *jedes* Eingabewort (hier natürliche Zahl) entschieden werden kann, ob die Zahl zur Menge gehört oder nicht. Der Algorithmus bricht also in jedem Fall in endlicher (vielleicht langer) Zeit ab. Die Turing-Maschine stoppt auf jeden Fall. Dies war die Definition von einer rekursiven Menge. Selbstverständlich ist das Komplement einer rekursiven Menge wiederum eine rekursive Menge. Um das zu sehen, brauchen wir aus einer Turing-Maschine T nur eine zweite T' so zu konstruieren, dass die akzeptierenden mit den nicht akzeptierenden Zuständen ausgetauscht werden. Wenn die Maschine T eine rekursive Menge M akzeptiert, so akzeptiert T' die komplementäre Menge von M.

Gibt es aber Mengen, welche zwar rekursiv aufzählbar aber nicht rekursiv sind? Oder anders: Sind rekursiv aufzählbare Mengen dasselbe wie rekursive Mengen?

Um diese Frage zu beantworten, beantworten wir zunächst eine andere Frage:

Gibt es auch Mengen, die nicht rekursiv aufzählbar sind? Ja, solche Mengen gibt es. Um das zu sehen, numerieren wir zunächst alle Turing-Maschinen. Dies ist möglich, da die Menge der Turing-Maschinen abzählbar ist. Die i-te Turing-Maschine wird mit T_i bezeichnet. Mit x_i bezeichnen wir ein beliebiges Wort aus einer Sprache. Dann ist die Wortmenge

$L_1 = \{ x_i \mid x_i$ wird von der Turing Maschine T_i nicht akzeptiert $\}$

nicht rekursiv aufzählbar, d.h. L_1 kann von keiner Turing-Maschine akzeptiert werden. Denn angenommen, dies wäre der Fall, dann haben wir eine Turing-Maschine T_j, welche L_1 akzeptierte; dann wäre $x_j \square L_1$ genau dann der Fall, wenn x_j nicht von T_j akzeptiert würde (nach Definition von L_1). Wir haben also

einen Widerspruch. Daher ist L_1 nicht rekursiv aufzählbar (Manne S. 39).

Jetzt kommen wir zurück auf die erste Frage: Sind rekursive und rekursiv aufzählbare Mengen dasselbe? Betrachten wir die Wortmenge

$$L_2 = \{ x_i \mid x_i \text{ wird von der Turing-Maschine } T_i \text{ akzeptiert } \}$$

Diese Menge ist offensichtlich rekursiv aufzählbar, da eine Turing-Maschine konstruiert werden kann, welche alle Wörter von L_2 akzeptiert. Denn jedes Wort aus $\sum^*$ ($\sum$ sei ein beliebiges finites Alphabet) wird in einer lexikographischen Ordnung numeriert als x_1, x_2, x_3, ... Aus dem i-ten Wort kann dann die T_i Turing-Maschine konstruiert werden. T ist so konstruiert, dass sie T_i simulieren kann. Daher akzeptiert sie das Wort x_i genau dann, wenn T_i auch x_i akzeptiert. Das Komplement von L_2 ist aber nichts anderes als die oben definierte Menge L_1. L_2 kann also nicht rekursive sein, da das Komplement einer rekursiven Menge ebenfalls rekursiv ist.
Wir wollen nochmals die Resultate zusammenfassen:

1. Wenn eine Menge rekursiv ist, so ist es auch ihr Komplement.
2. Eine Menge ist genau dann rekursiv, wenn sie und ihr Komplement rekursiv aufzählbar sind.
3. Die Klasse der rekursiven Mengen ist eine echte Unterklasse der rekursiv aufzählbaren Mengen.

Rekursiv aufzählbare Mengen könnte man auch als semi-entscheidbare Mengen bezeichnen, da es für sie eine *Prozedur* (aber keinen Algorithmus) gibt, denn sie terminiert unter Umständen nicht. Für die rekursiven Mengen gibt es jedoch einen *Algorithmus*, der nach Definition immer terminiert.

FUNKTIONEN

Jetzt wollen wir die von der Turing-Maschine *berechenbaren* Funktionen untersuchen. Dabei wollen wir nur Funktionen betrachten, die auf natürliche Zahlen definiert werden: y=f(x_1,x_2,..). Eine Funktion heisst **total**, wenn für alle x_1,x_2,.. Werte der Funktionswert definiert ist, sonst heisst die Funtion **partiell**.
Zu den rekursiv aufzählbaren Mengen und den rekursiven Mengen gibt es entsprechende Funktionsklassen: die **partiell rekursiven Funktionen** und die **(total) rekursiven Funktionen**. Partiell rekursive Funktionen ist genau die Klasse von Funktionen, welche von der Turing-Maschinen (als Generator) berechenbar sind, sodass die Turing-Maschine bei einer gegebenen Eingabe anhält oder eben auch nicht. Total rekursive Funktionen können auf Turing-Maschinen berechnet werden, die immer anhalten.

Die Funktionsklasse soll nun systematisch konstruiert werden.

(A) Folgende Funktionen heissen **Ausgangsfunktionen**

(1) Die 0-stellige Null-Funktion: f=0

(2) Die 1-stellige Nachfolgerfunktion: f(x)=x+1

(3) Die n-stellige Projektionsfunktion: $f_j^n(x_1,x_2, ...) = x_j$.

(B) Das **Kompositionsschema** (wobei $\boldsymbol{x}$ anstelle von x_1,x_2, ... steht):

(4) f($\boldsymbol{x}$) = g($f_1(\boldsymbol{x})$, ... , $f_m(\boldsymbol{x})$)

(C) Das **Rekursionsschema**:

(5) f($\boldsymbol{x}$,0) = g($\boldsymbol{x}$) und f(y+1) = h($\boldsymbol{x}$,y,f($\boldsymbol{x}$,y))

Die Funktionsklasse, welche mit Hilfe der Ausgangsfunktionen (1)-(3) den Kompositionsschema (4) und dem Rekursionsschema (5) konstruiert werden können, heisst die Menge der *primitiv-rekursiven Funktionen*. Diese Funktionsklasse ist sehr gross. So sind z.B. die vier arithmetischen Operationen der Addition, Subtraktion (wobei x-y=0 mit y>x), Multiplikation und Division primitiv-rekursiv. Die Addition beispielsweise kann ausgedrückt werden als (Engeler u.a S. 18):

add(x,0) = x
add(x,y+1) = add(x,y)+1

Die Menge der primitiv-rekursiven Funktionen ist klar abzählbar. Jede primitiv-rekursive Funktion ist auch total, wie man leicht einsehen kann. Die Menge der primitiv rekursiven Funktionen ist aber eine echte Untermenge der totalen Funktionen. Um dies einzusehen konstruiere man eine monoton steigende Funktion, die schneller wächst, als durch die Rekursion möglich ist. Ein solche ist die Ackermann Funktion. (Eine ausführliche Diskussion der Ackermann Funktion ist in Stetter S. 53ff zu finden). Die Ackermann Funktion ist also eine totale Funktion, die nicht primitiv rekursiv ist. Sie ist aber klar berechenbar, d.h. es gibt einen (Turing)-berechenbaren Algorithmus, welcher sie berechnet. Dies bedeutet aber, dass die Klasse der primitiv-rekursiven Funktionen nicht alle berechenbaren Funktionen enthält.

Eine weitere totale, nicht primitiv rekursive Funktion kann durch ein Diagonalisierungsargument gefunden werden: Es ist leicht einzusehen, dass die Menge der primitiv rekursiven Funktion abzählbar unendlich ist, d.h. man kann diese aussteigend der Menge N zuordnen:

$f_1, f_2, f_3, ...$

Sei $f(x)$ eine Funktion, die definiert ist als $f(x) = f_x(x)+1$. Diese Funktion ist

total, da sie für jedes $x \square N$ definiert ist. Wäre f in der Liste $f_1, f_2, f_3, \ldots$ vorhanden, so gäbe es ein j, sodass $f = f_j$. Wegen $f_j(j) = f(j)$ einerseits und $f(j) = f_j(j) + 1$ andererseits haben wir einen Widerspruch. Die Annahme, dass die Menge der totalen Funktionen abzählbar ist, führt daher zu einem Widerspruch. Die Menge der totalen Funktionen ist daher nicht abzählbar.

Wir erweitern die primitiv-rekursiven Funktionsklasse um ein weiteres Konstruktionsschema

(D) Das **μ–Schema**:

(6) $f(\boldsymbol{x}) = (\mu y)\,[h(\boldsymbol{x},y) = 0]$

welches zu einer gegebenen partiellen (n+1)-stelligen Funktion h die n-stellige Funktion f erzeugt, wobei $f(\boldsymbol{x})$ die kleinste Zahl y ist, für welche $h(\boldsymbol{x},y)=0$ wird, falls eine solche existiert. Eine typische Anwendung des μ-Operators ist die Konstruktion von Umkehrfunktionen. Der Logarithmus $y=\ln(x)$ z.B. ist definiert als die kleinste y Zahl, welche zu einer gegebenen Zahl x die Gleichung $e^y - x = 0$ erfüllt. Also haben wir

$\ln(x) = (\mu y)\,[e^y - x = 0]$

Die Funktionsklasse, welche (in endlichen Schritten) aus den obigen Ausgangsfunktionen (1)-(3) und den drei Konstruktionsschemata (4)-(6) konstruiert werden kann, heisst *μ–rekursiv*. μ-rekursive Funktionen sind offenbar ebenfalls berechenbar, denn entsteht $f(x)$ durch ein μ-Schema aus $h(\boldsymbol{x},y)$, so kann $f(x)$ durch folgende (Pseudo-Pascal)-Funktion berechnet werden:

```
function f(x:ARRAY): cardinal;
var y:cardinal;
begin
  y:=0;
  while H(x,y) ≠ 0 do y:=y+1;
  f := y;
end;
```

Allerdings kann nicht mehr garantiert werden, dass eine solche Funktion terminiert. Ist nämlich $h(\boldsymbol{x},y)\neq 0$ für alle $y\geq 0$, so terminiert die Funktion f nicht.

Man kann nun folgendes Theorem zeigen (ein Beweis findet sich in Engeler u.a. S. 23ff, oder Stetter S. 60):

Eine Funktion ist genau dann partiell-rekursiv, wenn sie μ-rekursiv und partiell ist; und eine Funktion ist genau dann total-rekursiv, wenn sie μ-rekursiv und total ist.

Damit haben wir ein systematisches Konstruktionsverfahren für alle (von Turing-Maschinen) berechenbaren Funktion. Gibt es auch nicht-berechenbare

Funktionen? Sicher, wie es nicht-rekursiv-aufzählbare Mengen gibt, gibt es auch nicht-partiell-rekursive Funktionen: Die totale Funktion, die oben durch das Diagonalisierungsverfahren konstruiert wurde, ist eine solche nicht-berechenbare Funktion. Man könnte es auch so ausdrücken: Da die Menge der berechenbaren Funktionen abzählbar ist – denn für jede solche Funktion kann eine Turing-Maschine konstruiert werden – die Menge aller Funktionen aber überabzählbar ist – eine Funktion ist nämlich eine Abbildung der Menge N in die Potzenzmenge 2^N – , muss es nicht-berechenbare Funktionen geben. Wir sehen hier wiederum, wie entscheidend das Diagonalisierungsargument von Cantor für die Berechenbarkeittheorie ist. Man könnte es auch noch anders ausdrücken: Die Menge der Funktionen ist grösser als die Menge der Funktionsnamen, d.h. es gibt mehr Funktionen als wir ihnen Namen vergeben können!

??. COMPUTER-SPRACHEN

...Dieses Kapitel noch zu schreiben... .
1-TM, = TM = PASCALLI = PASCALLINO = RAM = PART (Wagnar)
LOOP = partiel rekursiv; WHILE = TM = entscheidbar (Schöning).

6. UNENTSHEIBARE PROBLEME

Es könnte eine ganze Reihe von unentscheidbaren Problemen vorgestellt werden. Einige sind bereits in der Tabelle 1 zusammengestellt worden. Es sollen hier zwei weitere behandelt werden.

1. DAS HALTEPROBLEM

Minsky's Argumentation zum Halteproblems (1967, S.148-150). Jede Turing-Maschine TM, wie diese bisher beschrieben wurde, ist eine Maschine, die genau ein Problem lösen kann, nämlich entscheiden, ob ein Eingabewort zu einer Sprache gehört oder nicht; für eine andere Sprache muss eine neue Turing-Maschine konstruiert werden. Digital-Komputer, wie wir sie heute kennen, können aber allerlei Probleme lösen, da man sie programmieren kann; man braucht diese nicht für einen bestimmten Zweck neu zu konstruieren. Widerspricht diese Tatsache nicht der gemachten Aussage, dass jeder Digital-Computer durch eine Turing-Maschine simuliert werden kann? Oder brauchen wir dazu unendlich viele Turing-Maschinen, jede für eine bestimmten vorgegeben Zweck? Diesem Einwand kann durch das Konzept der **universellen Turing-Maschine** entgegnet werden. Jede für einen bestimmten Zweck konstruierte Turing-Maschine kann durch ihre Maschinen-Tafel

beschrieben werden. Diese Maschinen-Tafel kann selbst als ein Wort einer Sprache betrachtet werden; nennen wir es P. Das Eingabewort zu dieser speziellen Turing-Maschine nennen wir E. Betrachten wir nun die Konkatenation (getrennt durch ein Spezialzeichen, z.B. #) dieser beiden Wörter P und E, also P#E. Man könnte nun eine (universelle) Turing-Maschine so konstruieren, dass sie das Eingabewort P#E erhält und so funktioniert, dass sie die spezielle Turing-Maschine mit der Maschinen-Tafel P und dem Eingabewort E *simuliert*. Diese universelle Turing-Maschine hätte ihre eigene Maschinen-Tafel und würde ihren Lese/Schreibkopf zwischen dem linken Eingabewort (P), wo die Befehle für die spezielle Turing-Maschine 'abgespeichert' sind, und dem rechten Eingabewortteil (E) sich hin und her bewegen, um die Funktionsweise der speziellen Turing-Maschine zu simulieren. Es ist nicht schwierig einzusehen, dass ein solche universelle Maschine konstruiert werden kann, die so alle speziellen Machinen simulieren kann. (Penrose gibt die Maschinen-Tafel einer solchen Maschine an, S. 93). Diese universelle Turing-Maschine ist dann sehr ähnlich unseren *programmierbaren* Digital-Computern. Es ist unnötig zu sagen, dass eine solche Maschine sehr ineffizient arbeiten würde. Aber das spielt für unsere Überlegungen der Berechbarkeit keine Rolle.

Jede spezielle Turing-Maschine TM_x - wie gesagt - kann als Tupel (P_x,E_x) kodiert (gödelisiert) werden, wobei P_x die endliche Beschreibung der Übergänge (das Programm, welches die Turing-Maschine beschreibt, also ihre Maschinen-Tafel) und E_x die Bandbeschreibung am Anfang der Programmausführung, also die Eingabe zur Turing-Maschine TM_x, ist. Dieses Tupel kann einer universellen Maschine TM_u als Eingabe $P_x\#E_x$ auf das Band geschrieben werden, welche dann die Funktionsweise der Maschine TM_x simulieren kann. Wie diese Kodierung aussieht, spielt hier keine Rolle. Man kann sich das Tupel als zwei positive Zahlen vorstellen oder als zwei Zeichenfolgen bestehend aus einem beliebigen, endlichen Alphabet. Zahlentupel können bekanntlich durch das Diagonalverfahren von Cantor in eine einzige ganze Zahl abgebildet werden, sodass schliesslich jede Turing-Maschine zusammen mit ihrer Eingabe als eine ganze, positive Zahl kodiert werden kann.
Wir wollen nun zeigen, dass es keine (universelle) Turing-Maschine geben kann, die entscheidet, ob eine andere Turing-Maschine mit einer beliebigen Eingabe hält oder nicht. Der Beweis wird durch Widerspruch geführt.
Angenommen, TM_a sei eine (universelle) Turing Maschine, welche entscheiden

kann, ob eine beliebige Turing Maschine TM_x mit der beliebigen Eingabe E_x - also mit der Kodierung (P_x,E_x) - hält oder nicht. TM_a muss also auf jeden Fall entweder in einem akzeptierenden 'J' oder nicht-akzeptierenden Zustand 'N' halten. Solange sie nämlich nicht hält, kann gar nichts entschieden werden: Es könnte ja sein, dass sie in den nächsten 10 Stunden in einen akzeptierenden oder nicht-akzeptierenden Haltezustand übergeht. Aber solange sie läuft, kennen wir den Ausgang nicht. Das Band von TM_a werde also mit der Kodierung $P_x\#E_x$ beschrieben. Wir haben dann die Maschinenkodierung von TM_a gegeben als $(P_a,P_x\#E_x)$. Die Maschine TM_a mit der Eingabe $P_x\#E_x$ wird also mit der Ausgabe 'J' anhalten, genau dann, wenn die Maschine TM_x mit der Eingabe E_x hält; TM_a wird halten mit der Ausgabe 'N' genau dann, wenn TM_x mit der Eingabe E_x nicht hält.
Wenn die oben beschriebene Maschine TM_a mit der Kodierung $(P_a,P_x\#E_x)$ konstruiert werden kann, so kann sicher auch die Maschine mit der Kodierung $(P_a,P_x\#P_x)$ konstruiert werden, da P_x ja nichts anderes ist als eine spezieller Eingabekode zur Maschine TM_x, oder: da E_x beliebig sein kann, kann E_x sicher auch P_x sein. (Merke: P_x und E_x sind beide natürliche Zahlen). Die Maschine TM_a mit der Eingabe $P_x\#P_x$ hält also genau dann mit 'J', wenn die Maschine TM_x mit der Eingabe P_x hält. Wenn die Maschine TM_x mit der Eingabe P_x jedoch nicht hält, so hält die Maschine TM_a mit der Eingabe $P_x\#P_x$ mit der Ausgabe 'N'.
Es soll nun eine zur Maschine TM_a ähnliche zweite Maschine konstruiert werden, die wir TM_b nennen. Ihre Kodierung ist (P_b,E_b). TM_b sei identisch mit TM_a mit einer Ausnahme: Die Maschine TM_b soll in einer unendlichen Schlaufe hängen - also nicht halten, genau dann wenn TM_a mit der Ausgabe 'J' hält, und TM_b soll halten genau dann, wenn TM_a mit der Ausgabe 'N' hält. Da die Maschine TM_a nach Voraussetzung immer halten muss, haben wir nur diese beiden Möglichkeiten. Eine Maschine mit der genannten Modifikation kann leicht aus einer andern Turing-Maschine konstruiert werden: füge einfach am Ende der Programmbeschreibung, die zu einem 'J'-Zustand führt und dann hält, die Instruktionen hinzu, welche die Maschine in eine unendliche Schlaufe schicken. *Falls daher TM_a existiert, kann also auch TM_b leicht daraus konstruiert werden.*
Die Maschine TM_b mit der Eingabe $P_x\#E_x$ hält also genau dann, wenn die Maschine TM_x mit der Eingabe E_x nicht hält, und hält genau dann nicht, wenn die Maschine TM_x mit der Eingabe E_x hält.

Da nun TM_b ihrerseits eine Turing-Maschine ist, kann $P_b\#E_b$ selbst als ihre

Eingabe verwendet werden, d.h. wir können die Maschine mit dem Tupel (P_b,P_b#E_b) konstruieren. Daraus lässt sich aber ableiten, dass die Maschine TM_b mit der Eingabe P_b genau dann hält, wenn die Maschine TM_b mit der (beliebigen) Eingabe E_b nicht hält. Da E_b wie gesagt eine beliebige Eingabe sein kann, kann diese insbesondere auch P_b sein, sodass eine Maschine mit der Kodierung (P_b,P_b#P_b) erhalten wird. Für TM_b gibt es also eine Eingabe derart, dass TM_b genau dann hält, wenn TM_b nicht hält. Dies ist aber ein Widerspruch! Daraus folgt, dass eine solche Maschine TM_b nicht konstruiert werden kann, und dass daher auch TM_a nicht existieren kann, da TM_b aus TM_a konstruiert wurde. *Es gibt also keine Turing-Maschine derart, dass sie für jede Eingabe einer andern Maschine testen kann, ob jene Maschine hält oder nicht*. Das Halteproblem ist unentscheidbar.

2. DAS WORT-PROBLEM

Gegeben sein eine endliche Liste von 'Wortgleichheiten'. Wie zum Beispiel (Penrose S. 169ff)

```
EAT = AT            (Regel 1)
ATE = A             (Regel 2)
LATER = LOW         (Regel 3)
PAN = PILLOW        (Regel 4)
CARP = ME           (Regel 5)
```

Wir sagen das Wort links und rechts der Gleichheitszeichens seien 'gleich'. Die Wörter selber und die 'Gleichheit' brauchen nichts Spezielles zu bedeuten. Das Wortproblem besteht nun darin, aus dieser vorgegebenen Liste von Gleichheiten weitere Gleichheiten von andern Wörtern zu konstruieren. Die Konstruktionsvorschrift besteht darin, dass man ein Teilwort durch sein 'gleiches' ersetzen darf. Beispiel: Ausgehend von der obigen Liste soll gezeigt werden, dass die Wörter LAP und LEAP 'gleich' sind. Ihre Gleichheit kann gezeigt werden durch folgende Ableitung:

L**A**P = L**ATE**P (Anwendung der Regel 2)
L**AT**EP = L**EAT**EP (Anwendung der Regel 1)
LE**ATE**P = LE**A**P (Anwendung der Regel 2)

Die Frage ist nun: Können wir, wenn zwei beliebige Wörter gegeben sind, durch eine einfache Substitutionfolge (wie oben) vom eine Wort zum andern gelangen? Können wir also z.B. von CATERPILLAR zu MAN oder etwa von CARPET zu MEAT kommen? Im ersten Fall lautet die Lösung zufällig 'ja'. Im zweiten 'nein'. Wenn die Antwort 'ja' lautet, lässt sich eine solche Folge auch algorithmisch konstruieren. Im ersten Falle also duch die Folge

C**ATE**RPILLAR = C*A*RPILL**A**R = CARPIL**L*ATE*R** = CAR**PIL*LOW*** =

CAR*P* *AN* = *ME***A**N = M**E*** *AT E*N = M**A*****T*****E**N = M*A*N

Aber wie können wir sagen, dass es unmöglich ist von CARPET zu MEAT zu gelangen? Eine einfache Überlegung zeigt dies – der Weg zu dieser Überlegung ist jedoch alles andere als einfach: In jeder Gleichheit in der ursprünglichen Liste ist die Anzahl der Buchstaben A plus die Anzahl der Buchstaben W plus die Anzahl der Buchstaben M auf beiden Seiten gleich. Somit kann sich die Gesamtzahl dieser Buchstaben in Verlauf der Substitutionsequenz nicht ändern. Für CARPET ist diese Zahl 1, während für MEAT diese Zahl 2 beträgt. Daher gibt es keinen Weg, der von CARPET zu MEAT führt!

Das Wortproblem ist im allgemeinen unentscheidbar: Wenn zwei Wörter 'gleich' sind, so kann man dies durch einen Algorithmus zeigen, es gibt jedoch keinen Algorithmus, der entscheidet, ob zwei Wörter 'ungleich' sind!
Es ist jedoch bemerkenswert, dass ein findiger Mathematiker (hier Penrose) für unser Problem CARPET ≠MEAT eine Entscheidung herbeiführen konnte, obwohl man zeigen kann (was wir hier nicht getan haben), dass das Problem unentscheidbar ist! Können also menschliche Gehirne mehr als Algorithmen? Interessant ist aber, dass, wenn die erwähnte Summenregel einmal gefunden ist, sie in einen Algorithmus eingebunden werden kann! Aber für ihr Auffinden braucht es offenbar mehr als blosses mechanisches Ableiten, Penrose nennt es 'mathematische Einsicht' ('insight').....

7. GÖDEL'S UNVOLLSTÄNDIGKEITSSATZ

Am internationalen Mathematiker-Kongress 1928, erweiterte Hilbert sein formalistisches Programm. Er stellte drei Fragen:

1) Ist die Mathematik vollständig, d.h. kann jede mathematische Formel als wahr oder falsch bewiesen werden?
2) Ist die Mathematik konsistent, d.h. ist es nicht der Fall, dass ein falsche Aussage abgeleitet werden kann?
3) Ist die Mathematik entscheidbar, d.h. gibt es eine finite Methode (Algorithmus), welche für jede Formel entscheidet, ob er wahr oder falsch ist?

Hilbert war überzeugt, dass alle drei Fragen eine positive Antwort haben. Aber schon drei Jahre später, bewies Gödel seinen berühmten Unvollständigkeitssatz, dessen Beweis hier kurz skizziert werden soll (eine Rekonstruktion aus Nagel und Newman, 1958).

1931 konnte Gödel zeigen, dass die Arithmetik unvollständig sein muss, d.h. dass es mathematische Formeln gibt, die weder als wahr noch als falsch bewiesen werden können. Die Formalisierung der Gödelschen Beweisführung sowie die nachfolgende Klärung des Algorithmusbegriffs sind eine der grossen intellektuellen Errungenschaften dieses Jahrhunderts. Sie ist die theoretische Basis der Informatik.
Die Idee des Gödelschen Unvollständigkeitsbeweises besteht darin eine Formel zu konstruieren, die ihre eigene Unbeweisbarkeit behauptet. Die Gedankengänge sind ähnlich wie beim Halteproblem, welches aus moderner Sicht nichts anderes ist als der automaten-theoretische Ausdruck des Gödelschen Unvollständigkeitstheorems. Beim Halteproblem ging es darum, eine Turing-Maschine zu konstruieren, welche entscheiden kann, ob eine beliebige Turing-Maschine hält oder nicht. Als Ausgangspunkt des Beweises wurde angenommen, dass eine solche Maschine existiert. Die Maschine wurde dann mit einem Eingabekode gefüttert, welche die Maschine selbst - also sich selbst - 'repräsentiert'. Durch diesen *Selbstbezug* konnte dann gezeigt werden, dass es eine solche Maschine nur gibt, wenn es sie nicht gibt. Ein Widerspruch, woraus dann wohl abzuleiten ist, dass die Ausgangsannahme, nämlich dass eine solche Maschine existiert, falsch ist.

Der Gödelsche Beweis wird an einem reduzierten, formalen System - einem Teilsystem der Prädikatenlogik - durchgeführt, um die einzelnen Beweisschritte klar hervorzuheben. In diesem System gibt es nur neun Symbole:

$$\neg \vee \exists = 0\ s\ (\ ),$$

mit den entsprechenden prädikatenlogischen Interpretationen: 0 ist die Zahl Null, *s* ist die Nachfolgerfunktion, d.h. 1 wird als *s*0, 2 als *ss*0 (oder *s*1) usw. dargestellt. Jedem Symbol wird eine Zahl von 1 bis 9 in der obigen Reihenfolge zugeordnet: Also ¬ erhält die Zahl 1, ∨ die Zahl 2, ...,) die Zahl 8 und , (Komma) die Zahl 9. Individuelle Variablen $x,y,z,...$ erhalten verschiedene Primzahlen grösser 10 zugewiesen: also *x* erhält 11, *y* erhält die Zahl 13 und *z* die Zahl 17, usw. Jede Aussagenvariable $p,q,r,...$ erhält die Quadratzahl der Primzahlen grösser 10 zugewiesen: also *p* erhält 11^2, *q* die Zahl 13^2 usw. Jede Prädikatsvariable $P,Q,R,...$ erhält die Kubikzahl zugewiesen, also *P* erhält 11^3, *Q* erhält die Zahl 13^3 usw. Damit hat nun jedes individuelle Symbol im (reduzierten) prädikatenlogischen Kalkül eine ihm entsprechende, eindeutig zugewiesene Zahl: Die Zahl 5, z. B. entspricht dem Symbol =, die Zahl 13 dem Symbol *y*, die Zahl 15 entspricht keinem individuellen Symbol.

Eine prädikatenlogische Formel kann nun als Zahlenfolge kodiert werden, zum Beispiel erhält die Formel

$$(\exists x)(x = sy) \qquad (1)$$

die Zahlenfolge: 8, 4, 11, 9, 8, 11, 5, 7, 13, 9. Aus der Zahlenfolge wird eine eindeutige Zahl - genannt die Gödelzahl (GZ) - folgendermassen konstruiert: Die Primzahlen 2, 3, 5, 7, 11, 13,... werden mit den einzelnen Zahlen in der Folge exponiert und das Resultat wird dann multipliziert. Die Gödelzahl des Ausducks (1) ist somit

$$2^8 \cdot 3^4 \cdot 5^{11} \cdot 7^9 \cdot 11^8 \cdot 13^{11} \cdot 17^5 \cdot 19^7 \cdot 23^{13} \cdot 29^9$$

Dies entspricht der grossen Zahl:

145 666 408 661 709 409 197 789 938 288 649 818 781 891 470
181 481 887 898 950 349 321 995 516 094 737 500 000 000.

Jede Formel F kann so in eine eindeutige Gödelzahl überführt werden, die mit GZ(F) bezeichnet wird.

Die Gödelzahl der Formel (1) soll im folgenden *m* genannt werden. Die Zahl selber ist unwichtig, entscheidend ist, dass jede prädikatenlogische Formel eine eindeutige Gödelzahl besitzt, und dass aus einer Zahl auch wiederum die Formel durch Primzahlenfaktorisierung eindeutig reproduzierbar ist. Nicht jede Zahl entspricht einer syntaktisch legalen Formel. Dies ist aber unwichtig.

Ein Beweis ist nichts anderes als eine Sequenz von Sätzen: Soll eine Formel z bewiesen werden, so werden ausgehend von einer endlichen Anzahl Axiomen x_1, x_2,...,x_k weitere Sätze x_{k+1},x_{k+2},...,x_{k+n} abgeleitet, bis x_{k+n} der gewünschten Formel z entspricht, (oder ¬z in der Sequenz vorkommt, woraus dann zu schliessen ist, dass z nicht ableitbar ist). Auch einem Beweis kann eine eindeutige Zahl zugeordnet werden: Wenn der Beweis aus den Formel Sequenz x_1,...,x_n besteht, und GZ(x_1),..., GZ(x_n) die Gödelzahlen der einzelnen Formeln im Beweis sind, so erhält der Beweis die Gödelzahl

$$2^{GZ(x1)} \cdot 3^{GZ(x2)} \cdot \ldots \cdot p_n^{GZ(xn)}$$

wobei p_n die *n*-te Primzahl ist.

Es soll nun ein neues Prädikat DEM eingeführt werden. DEM besitzt zwei numerische Parameter x,z, also DEM(x,z). Die Bedeutung ist folgendermassen: Wenn x die Gödelzahl eines Beweises (einer Formelsequenz) und z die Gödelzahl einer Formel ist, und wenn z aus x abgeleitet werden kann, dann soll

das Prädikat DEM(x,z) wahr zurückgeben, in allen andern Fällen ist es falsch. DEM(x,z) kann also interpretiert werden als 'die Formelsequenz mit der Gödelzahl x ist ein Beweis für die Formel mit der Gödelzahl z'. Diese Aussage ist wahr genau dann, wenn die Formelsequenz mit der Gödelzahl x in der Tat ein Beweis für die Formel mit der Gödelzahl z ist, sonst ist die Aussage falsch.
Es soll eine weitere Funktion SUB mit drei numerischen Parametern (a,b,c) eingeführt werden, also SUB(a,b,c), mit der folgenden Bedeutung: Kommt in der Formel mit der Gödelzahl a ein Symbol mit der Zahl b vor, so soll dieses Symbol durch die Zahl c ersetzt (substituiert) werden. Die Funktion SUB(a,b,c) gibt die Gödelzahl der daraus resultierenden Formel zurück. Beispiel: SUB(m,13,m) bedeutet (m sei die Gödelzahl der oben angegebenen Formel (1)): in $(\exists x)(x = sy)$ soll 'y' durch die Zahl m ersetzt werden, sodass man $(\exists x)(x = sm)$ erhält. Da die Ziffern nicht zum Alphabet unseres Systems gehört, muss die Zahl m in der Formel durch *sssssss....s*0 ersetzt werden, wobei das Symbol *s* m+1-mal vorkommt. Man erhält somit die Formel $(\exists x)(x = ssssss...s0)$. Der Ausdruck SUB(m,13,m) gibt dann die astronomische Gödelzahl der Formel $(\exists x)(x = ssssss...s0)$ zurück.

Nun sind wir endlich in der Lage eine Formel G zu konstruieren, welche besagt, dass G nicht beweisbar ist, nämlich auf folgende Art:

G ist identisch mit ¬DEM(x,GZ(G)).

Das heisst, man muss eine Formel G finden, in welcher ihre eigene Gödelzahl vorkommt. Wenn dies gelingt, so haben wir gezeigt
1) dass G nicht beweisbar ist (Unvollständigkeit),
2) dass, falls G beweisbar ist, ¬G auch beweisbar ist (Inkonsitenz),
3) dass G, obwohl nicht beweisbar, trotzdem wahr ist.

Um die Formel G zu konstruieren, führe man zunächst folgende Formel (2) ein:

$$(\forall x)\neg DEM(x,z) \qquad (2)$$

welche besagt, dass für jedes x, die Formelsequenz mit der Gödelzahl x kein Beweis für die Formel mit der Gödelzahl z sei. Da ∀ kein Symbol unseres reduzierten Alphabetes ist, kann die Formel ersetzt werden durch

$$\neg(\exists x)DEM(x,z) \qquad (3)$$

Man beachte, dass z eine offene Variable ist, für die man nun eine konkrete Zahl, z.B SUB(y,13,y), einsetzen kann. Dann erhält man die Formel

$$\neg(\exists x)\mathrm{DEM}(x,\mathrm{SUB}(y,13,y)) \qquad (4)$$

Die Gödelzahl für die Formel (4) sei n. Da y eine weitere offene Variable in (4) ist, substituiere man die Zahl n und man erhält die gesuchte Gödel-Formel G

$$\neg(\exists x)\mathrm{DEM}(x,\mathrm{SUB}(n,13,n)) \qquad (5)$$

wobei n die Gödelzahl der Formel (4) sei.
Die Formel (5) entspricht nun ihrerseits einer Gödelzahl, die aber nichts anderes ist als SUB(n,13,n) selbst. Denn SUB(n,13,n) ist die Gödelzahl der Formel, welche aus der Formel mit der Gödelzahl n - nämlich Formel (4) - durch die Substitution von y erhalten wurde. Formel (5) aber wurde ebenso durch dieselbe Substitution erhalten. Also sind die beiden Gödelzahlen identisch. Damit haben wir die gewünschte Formel: G besagt nämlich, dass es kein x gibt (d.h. dass es keinen Beweis gibt), der die Formel der Gödelzahl SUB(n,13,n) (d.h. G) ableitet. G sagt also aus, dass G nicht ableitbar ist.

Nun kann die Frage gestellt werden, ob G ein Theorem (Satz) ist? Wenn ja, dann muss es aus den Axiomen des Systems ableitbar sein. Dies ist aber gerade nicht der Fall, da die Formel G besagt, dass G nicht ableitbar ist. Wenn wir also annehmen, dass G ein Theorem ist, dann ist G kein Theorem und es resultiert ein inkonsistentes System. Also ist G kein Theorem? Dann ist G in der Tat wahr, denn es besagt, dass G nicht ableitbar ist. Dann aber gibt es mindestens einen Satz - nämlich G - der nicht ableitbar ist, denn G besagt ja zu Recht gerade, dass G nicht ableitbar sei. Dann resultiert zwar ein konsistentes aber nicht vollständiges System. Wie dem auch sei, man muss die Konsequenz ziehen, dass das System nicht gleichzeitig konsistent und vollständig sein kann.
Wenn die Unvollständigkeit als das kleinere Übel gewählt wird (denn aus einem inkonsistenten System kann jede Formel abgeleitet werden), könnte man sich retten, indem G als ein Axiom ins System aufnimmt, um Vollständigkeit wieder herzustellen. Dann kann man aber mit einem ähnlichen Verfahren zeigen, dass das System immer noch unvollständig ist. Man müsste schon unendlich viele Axiome aufnehmen. Aber selbst dies genügte nicht, in ähnlicher Weise könnte wiederum ein mit G vergleichbarer Satz abgeleitet werden. Das System muss prizipiell unvollständig bleiben.

Zusammenfassung: Die Essenz des Gödelschen Beweises, wie wir gesehen haben, besteht darin, eine Formel in eine ganze Zahl (Gödelzahl) umzuwandeln, dann diese Zahl selbst in einer weiteren Formel - Zahlen sind ja syntaktische Bestandteile der Prädikatenlogik - einzusetzen, die besagt, dass die Formel zur eingesetzten Gödelzahl nicht ableitbar sei. Wenn nun die eingesetzte Zahl gerade der Gödelzahl der Formel entspricht, in welche die Zahl eingesetzt wird, erhält man die obige Formel G, welche dann – wie sie selbst sagt – nicht ableitbar ist.

Damit waren 1931 die ersten beiden Fragen von Hilbert (negativ) beantwortet. Wie steht es mit der dritten Frage von Hilbert, ob Mathematik entscheidbar sei. Gödel schloss nicht aus, dass es eine Methode geben könnte, welche entscheidbare von unentscheidbaren Sätzen unterscheidet. Aber fünf Jahre später (1936) wurde auch diese Frage durch die Konstruktion des Halteproblems auf der Turing-Maschine von Turing negativ beantwortet. (Siehe dazu auch die Turing Biographie von Hodges 1983, S.93-110).

Nachtrag: Man erinnert sich an das Paradox von Epimenides, der ein Kreter war, dass alle Kreter lügen. Wie man sich auch wendet, der Satz kann weder falsch noch wahr sein. Gödel könnte man sich als ein moderner Epimenides vorstellen, der sagte: “Dieser Satz ist unbeweisbar”. ('Dieser Satz' bezieht sich natürlich auf sich selbst!). Ist dieser Satz beweisbar? Wenn ja, dann ist er wahr. Der Satz besagt jedoch selbst, dass er unbeweisbar ist. Also haben wir einen Widerspruch. Dann muss er also unbeweisbar sein, was wiederum besagt, dass er wahr ist, da er ja seine eigene Ungeweisbarkeit besagt. Also haben wir einen wahren Satz, der unbeweisbar ist!

9. EINE NOTIZ ZUR INTELLIGENZ

If you treat machines like people,
you're likely to end up treating people like machines.

Haben Maschinen dieselbe Denkkraft wie Menschen? Der berühmte Turing-Test (Turing 1950) schreibt einer Maschine dann eine gewisse Intelligenz zu, wenn - grob gesprochen - nach einer bestimmten Zeit eine menschliche Testperson seinen 'Gesprächspartner' nicht als Maschine oder menschliche Person identifizieren kann. Der Turing-Test hat den Vorteil, dass er die Entscheidung über Intelligenz nicht irgendwelchen 'Wesenskriterien' über Intelligenz unterstellt, sondern dass er ein empirischer Test ist. Turing selber glaubte, dass bis Ende des 20igsten Jahrhunderts eine Maschine den Test bestehen könnte. Bisher sind wir jedoch noch weit davon entfernt, eine solche Maschine zu haben. Man kann ernsthaft daran zweifeln, dass es eine solche Maschine jemals geben wird. Die Vertreter der starken AI (siehe Penrose), wie Minsky, glauben, dass es genügt, den heutigen Maschinen nur genügend grosse Wissensbasen und Algorithmen zur Verfügung zu stellen, und Maschinen werden dann 'von alleine' intelligent. Andere, wie Penrose, glauben, dass es einen qualitativen, fundamentalen Schritt braucht, quasi einen Quantensprung.

(Penrose glaubt, dass diese Frage nur nach einer ganz neuartigen Quantenmechanik geklärt werden könnte). Vielleicht müssen Maschinen zunächst Menschinen werden, bevor sie auch nur annähenrd ein primitives Stadium von Intelligenz haben. Aber wir feilschen hier mit Worten!
Selbst, wenn Maschinen den Turing-Test bestehen, sind sie dann intelligent? Ist der Turing-Test ein adäquates Modell der Intelligenz? Der Turing-Test hinterlässt die etwas unbefriedigende Situation, dass es offenbar genügt, wenn eine Maschine Intelligenz *simuliert*, damit sie als intelligent gelten kann! Verschiedene – auf den ersten Blick verblüffende – Programme wurden ja geschrieben, welche die Antworten und Fragen eines Gesprächspartners (z.B. eines Psychiaters) simulieren, um damit eine gewisse Intelligenz *vorzutäuschen*. Solche Programme haben meiner Meinung nach nicht das geringste mit Intelligenz zu tun, nicht weil die Maschine simuliert – Menschen simulieren ja auch manchmal, sondern weil sie *nur* simulieren kann. Gewiss, woran soll man erkennen, dass jemand simuliert? Es gibt keine prinzipielle Methode, die entscheiden könnte, dass jemand 'nur' simuliert. Einer Maschine, die während ihrer ganzen Existenz nur immer vorgibt intelligent zu sein und 'wir' merken es nicht, muss man offenbar Intelligenz zuerkennen. Es ist allerdings wiederum zweifelhaft, ob es eine solche Maschine je geben kann.

Intelligenz wurde ja auch oft mit Schachkomputer in Verbindung gebracht. Wäre ein Schachkomputer, welcher Grossmeister schlagen könnte – dies ist heute schon (fast) Realität – als intelligent zu bezeichnen? Solchen Maschinen wäre ich schon eher geneigt eine gewisse Intelligenz zu zuschreiben. Innerhalb des Schachuniversums mögen solche Maschinen unschlagbar sein, sodass sie nur noch gegeneinander antreten könnten. Gewiss sind heutige Computer auch unschlagbar zur Ausführung einer Addition. Aber im Gegensatz zum Schach, ist die Addition eine rein mechanische Ausführung von Instruktionen. Mechanisches Abarbeiten (Algorithmen) ist aber gerade das Gegenteil von Intelligenz! Aber ein Schachprogramm arbeitet ja auch rein mechanisch seine Instruktionen ab, also wo ist der Unterschied? Auf dieser Ebene könnte man allerdings entgegnen, dass im Gehirn, wenn wir die 'Ausführung' physiologisch verfolgen würden, die 'Instruktiuonen' ebenfalls 'mechanisch' (mindestens quanten-mechanisch wie im Computer-Prozessor auch) abgearbeitet werden. Im Schachprogramm sind aber *mehrere* Algorithmen und Methoden eingebaut, zwischen denen das 'Hauptprogramm' heuristisch auswählen kann. Ja aber können denn mehrere Algorithmen mehr als ein Algoritmus? *Genau das ist es, was ich behaupte.*

Betrachten wir das obige Wortproblem. Wir wissen, dass das Problem unentscheidbar ist, d.h. dass es keinen Algorithmus gibt, der alle Wortprobleme löst. Jedoch gibt es für verschiedene Wortprobleminstanzen verschiednene, möglicherweise ganz unterschiedliche Algorithmen, welche die Instanz lösen, wie oben die Anwendung der Additionsregel für das ganz spezielle Wortproblem. Viele Algorithmen können damit viele Wortprobleminstanzen lösen. Kennt man diese Algorithmen einmal, so kann man eine ganze Reihe von speziellen Wortproblemen mechanisch lösen. (Man beachte, dass das allgemeine Wortproblem unlösbar ist!) Wie aber findet man diese Algorithmen? Einen Algorithmus zu finden ist selber ein unlösbares Problem. Man verfährt hier gleich wie bein (unlösbaren) Wortproblem: Man wählt ein bereits bekanntes Verfahren oder man wandelt ein bekanntes Verfahren leicht oder radikal ab und fügt es den bekannten Verfahren hinzu. So entsteht mit der Zeit eine erweiterbare Methodenbank. Um den entsprechenden Algorithmus aus der Methodenbank auszuwählen, braucht es eine Metaheuristik, die wiederum in einem oder einer ganzen Reihe von Algorithmen kodiert werden kann. So entsteht eine ganze Hierachie von Algorithmen und Meta-Algorithmen, die mehr oder weniger intelligent einander aufrufen können. Dies bedeutet nicht, dass das bunte Zusammenmischen von Algorithmen und Meta-Algorithmen plötzlich intelligentes Verhalten entsteht, es braucht dazu eine bestimmte *Ordnungsstruktur*. In biologischen Systemen ist diese Ordnungsstruktur durch die Evolution entstanden. In diesem Sinne, glaube ich, besteht eine bestimmte Plausibilität für die These der Exponenten der starken AI.

Die Tatsache, dass es unentscheidbare Probleme gibt, die also von einem Algorithmus nicht gelöst werden können, hat verschiedene Forscher dazu bewogen, gerade darin einen fundamentalen Unterschied zwischen der menschlichen und der künstlichen Intelligenz zu konstruieren. Ich glaube, dass dieser Unterschied nur ein gradueller ist. Diese Forscher argumentieren folgendermassen (Siehe Lucas 1961):
Gerade Gödel's Unvollständigkeitstheorem zeigt, dass es einen fundamentalen Unterschied zwischen Menschen und Maschinen gibt. Denn Gödel zeigte, dass es in einem formalen System, das genügend mächtig ist um die Arithmetik zu umfassen, Sätze gibt, die im System selber nicht bewiesen werden können. Wir Menschen können aber offenbar feststellen, dass es solche Sätze gibt, und dass solche formalen Systeme nicht vollständig sind, d.h. wir Menschen sind fähig

'ausserhalb' des Systems zu stehen, wozu Computer offenbar nicht fähig sind, da es das 'Wesen' einer Maschine ist, in einem formalen System eingebettet zu sein, was nichts anderes heisst, als dass sie konsistent und logisch aufgebaut sein muss. Also kann ein Computer nicht fähig sein, Formeln, die nicht bewiesen (abgeleitet) werden können, als wahr zu erkennen. Da wir Menschen diese aber erkennen können, der Computer aber nicht, folgt daraus, dass der Computer nie ein adäquates Modell des menschichen Geistes sein kann. Lucas' Schlussfolgerung: "...minds are essentially different from machines". (zitiert nach Narayanan, S.44). Dieselbe Argumentation wird auch von Penrose vertreten (S. 181). Penrose geht allerdings noch weiter und postuliert einen quanten-physikalischen Bezug, welche das Gehirn offenbar haben soll, die Maschine aber nicht.

Das eigentliche Argument von Lucas besteht also darin, dass Maschinen nicht *aus dem System springen* können, um zu reflektieren, was sie tun. Das ist sicher richtig für die heutigen Maschinen, und in diesem Sinne werden heutige Maschinen, die 'einfach' einen Algorithmus ausführen, nie fähig sein, den Algorithmus zu 'reflektieren'. Aber vielleicht sind es die Menschinen (eine Art Nachfolger von den heutigen Maschinen)! Eine primitive Art, einen Algorithmus zu 'reflektieren' ist beispielsweise – nach einer Zeitspanne des Misserfolgs mit einem ersten Algorithmus – einfach einen andern auszuprobieren. Wenn wir Menschen beispielsweise mit einem Verfahren nicht ans Ziel gelangen, so versuchen wir es oft einfach damit, dass wir das ganze Verfahren abbrechen, und ein anderes versuchen. Dies führt dann vielleicht zum Erfolg – vielleicht auch nicht. Aber dies *wäre* eine (allerdings primitive) Art des 'aus-dem-System-Springen'. Ich sage nicht, dass durch blindes Ausprobieren systematisch Probleme gelöst werden können. Die biologische Evolution konnte ja auch nicht einfach dadurch fortschreiten, indem die Gene (oder – um noch eine Stufe 'tiefer' zu gehen – die Atome) sich blind zufällig permutieren und verbinden. Es brauchte dazu noch den Druck der Selektion! (siehe dazu das faszinierende Buch von Dawkins).

Ist der Ableitungsweg des Gödelschen Satzes einmal bekannt, dann kann man ihn auch rein mechanisch nachvollziehen! Es besteht kein Grund anzunehmen, was das Unvollständigkeitstheorem selber von einer Maschine nicht abgeleitet werden könnte, wenn er nur genügend 'Hintergrundwissen' enthält.

Ein weiterer Argumentationsstrang der Befürworter, dass Maschinen und

Menschen prinzipiell unterschiedlich seien, geht so (Narayanan S. 67): Turing-Maschinen sind die mächtigsten (im Sinne der Berechbarkeit), bekannten Maschinen wie wir ja aus der Berechbarkeitstheorie wissen, und es besteht wenig Hoffnung, dass noch mächtigere Maschinen einmal erfunden werden. Da aber eine Turing-Maschine (TM) nicht alle möglichen Funktionen berechnen kann (da es von diesen überabzählbare viele gibt, eine TM aber nur abzählbar viele berechnen kann), und da insbesondere selbst für korrekte Turing-Programme nicht entschieden werden kann, ob die TM auf diesen Programmen hält oder nicht (Halteproblem), kann die TM selber nicht 'sehen', ob sie in einer unendlichen Schaufe hängt oder ob das Programm, auf dem die TM gerade arbeitet, nach einer endlichen Zeit abbricht. Turing-Maschinen können also in eine unendliche Schlaufe hineingeraten und nie mehr 'herausfinden'. Da wir Menschen aber nicht in eine unendliche Schlaufe geraten können, wenn wir dieselbe Aufgabe bewältigen, folgt daraus, dass wir keine TM's sind. ("Since we humans do not go into an eternal loop when tackling the same problems, it follows that we are not TMs. But TMs are considered to be the most powerful type of computer imaginable. Hence, we humans cannot be machines, since we can do something more (i.e. not go into a loop) than even the most powerful machine imaginable.") (p.67).

Diese Argumentation scheint auf den ersten Blick sehr tiefsinnig zu sein, da sie auf den eigentlichen Grenzen der mechanischen Berechenbarkeit beruht. Aber hier wird die physikalische 'Implementation' eines Programm mit der abstrakten 'Spezifikation' der TM verwechselt: Der Mensch kann ganz einfach nicht in eine unendliche Schlaufe geraten, weil er nicht ewig lebt! Zudem gibt es nichts einfacheres als zu garantieren, dass ein (physisches) Programm hält: Man integriert einen Zeitzähler ins Programm selber. Nach einer bestimmten Zeit muss die Abarbeitung fertig sein oder das Programm verabschiedet sich von selbst! Jeder Programmierer weiss wie das zu machen ist!

Aber selbst wenn der Mensch ewig wäre, wäre die Argumentation falsch, dass 'humans do not go into an eternal loop'. Betrachten wir die Vermutung von Fermat. Diese Vermutung ist seit 3 Jahrhunderten ein offenes mathematisches Problem. Es gibt verschiedene, künftige Entwicklung zu diesem Problem:

1) Die Vermutung kann bewiesen werden.
2) Die Vermutung kann widerlegt werden.
3) Es kann gezeigt werden, dass die Vermutung weder bewiesen noch widerlegt werden kann (Es ist wie das Halteproblem ein unlösbares Problem).
4) Die Vermutung bleibt weiterhin offen[1].

Trifft 1, 2 oder 3 zu, so findet das Problem eine 'abschliessende' Antwort, trifft hingegen 4 zu, so geht die (menschliche) Forschung bezüglich der Fermat'schen Vermutung in eine ewige Schlaufe. Also auch Menschen können in einer 'ewigen' Schlaufe hängenbleiben!

Kurios genug ist, dass selbst für Probleme, die längst gelöst sind, die menschliche Suche in eine 'unendliche' Schlaufe gehen kann: Es ist zum Beispiel längst bekannt, dass die Dreiteilung eines Winkels mit Zirkel, Lineal und Bleistift unmöglich ist. Trotzdem gibt es immer wieder einige Verwegene, welche eine Konstruktion mit Zirkel und Lineal versuchen! So werden sie sich wohl sagen, dass die Fachleute gut Reden haben: alle Konstruktionen werden diese wohl nicht ausprobiert haben können, und so könnte es doch vielleicht sein, wenn nur die Sterne günstig stehen, dass eine Konstruktion doch noch gelingen könnte!
(Zum Problem der Dreiteilung eines Winkels siehe Tietze S. 61).

(1) Gerade dieser Tage machte die sensationelle Meldung die Runde, dass die Fermatsche Hypothese von einem Mathematiker gewiesen worden sei. Dies ändert an meiner Argumentation nicht das geringste. Falls es den Leser stört, ersetze er einfach jeden Bezug auf die Fermatsche Vermutung durch einen Bezug auf die Goldbach'sche Vermutung.

Hat ein Hund Buddha-Natur?
Das ist die ernsteste von allen Fragen.
Sagst du ja oder nein,
verlierst du deine eigene Buddha-Natur.
Hofstadter, GEB, S.294.

REFERENCES

ARBIB M.A., [1987], Brains, Machines, and Mathematics, 2nd edition, Springer.

DAVIS M.D., WEYUKER E. J., [1983], Computability, Complexity, and Languages, Fundamentals of Theoretical Comptuer Science, Academic Press, New York.

DAWKINS R., [1987], The Blind Watchmaker, W.W. Norton & Comp, New York.

GÖDEL K., [1931], Über formal unentscheidbare Sätze der Principia mathematica und verwandte Systeme, Monatshefte für Mathematik und Physik, 38, p.173-198.

HODGES A., [1983], Alan Turing, the enigma, Vintage paperback edition 1992.

HERSCHEL R., [1974], Einführung in die Theorie der Automaten, Sprachen und Algorithmen, Oldenbourg.

HOFSTADTER D.R., [1985], Gödel, Escher, Bach, ein Endloses Geflochtenes Band, Klett-Cotta, Stuttgart (übersetzt vom engl. 1979).

HOPCROFT J.E., ULLMAN J.D., [1990], Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie, Addison-Wesley. (engl. 1979).

LEWIS H.R., PAPADIMITRIOU C.H., [1981], Elements of the Theory of Computation, Prentice-Hall.

LINZ P, [1990], An Introduction to Formal Languages and Automata, D.C.Heath and Comp., Lexington.

LUCAS J.R., [1961], Minds, machines and Gîdel, Philosophy, 36, reprinted in: ANDERSON, A.R. (ed), Minds and machines. Prentice Hall, 1964.

MANNA Z. [1974], Mathematical Theory of Computation, McGraw-Hill Book Company, New York.

MANDRIOLI D., GHEZZI C., [1987], Theoretical Foundations of Computer Science, Wiley.

MINSKY M.L., [1967], Computation: finite and infinite machines, Prentice-Hall.

NAGEL E., NEWMAN J.R. [1958], Gödel's proof. New York University Press.

NARAYANAN A., [1988], On being a machine, Vol 1, Formal Aspects of Artififial Intelligence, Ellis Horwood Limited.

PENROSE R., [1989], The emperor's new mind: Concerning Computers, Minds, and the Laws of Physics, Oxford University Press.

SIELAFF K., [1971], Einführung in die Theorie der Gruppen, Diesterweg Salle.

STETTER F., [1988], Grundbegriffe der theoretischen Informatik, Springer.

TIETZE H., [1982], Gelöste und ungelöste Probleme aus alter und neuer Zeit, Bd 1, dtv wissenschaft. (erste Ausgabe 1959).

TURING A.M., [1950], Computing machinery and intelligence, Mind, LIX, p.433-460.

TURING A.M., [1936], On Computable numbers, with an application to the Entscheidungsproblem, Proc. London Math. Soc., Ser. 2 Vol 42, p230-265.

BALKE L., BÖHLING K.H., [1993], Einführung in die Automatentheorie und Theorie formaler Sprachen, Reihe Informatik Bd. 90, BI Wissenschaftsverlag, Mannheim.

Theoretisch orientiert sich an der Chomsky-Hierarchie. Konzentriert sich auf Typ1 - Typ 3 Sprachen.

FLOYD R.W., BEIGEL R., [1994], The Language of Machines, An Introduction to Computability and Formal Languages, Computer Science Press, W.H.Freeman and Comp., New York.

Ausführliche, systematische Einführung. Vereinheitlichung der Maschinen und Sprachen. Könnte ein Standardwerk werden.

SCHÖNING U., [1992], Theoretische Informatik kurz gefasst, BI Wissenschaftsverlag, Mannheim.

Kurze sehr lesbare Einführung in formale Sprachen, Berechenbarkeit und Komplexität. LOOP- und WHILE-.Sprache wird eingeführt.

WAGNER K.W., [1994], Theoretische Informatik, Gundlagen und Modelle, Springer-Lehrbuch, Berlin.

Macht die Äquivalenz der Berechenbarkeiten von PASCALLI (einem Subset von Pascal), RAM, TM plausible, gibt dazu die Übersetzungsregeln an. Sehr lesbar und empfehlensswert.