Theoretische Informatik

Reinhard Völler

2. März 1999

Inhaltsverzeichnis

1 Algorithmen, Berechenbarkeit, Entscheidbarkeit **1**
1.1 Algorithmen . . . 2
1.2 Eigenschaften eines Algorithmus . . . 4
1.3 Aufzähl- und Entscheidungsverfahren . . . 6
1.4 Berechenbare Funktionen . . . 7
1.5 Abzählbarkeit . . . 8
1.6 Entscheidbare Mengen . . . 10
1.7 Charakteristische Funktionen . . . 11
1.8 Aufzählbare Mengen . . . 12

2 Turing-Maschinen **17**
2.1 Das Maschinenmodell . . . 18
2.1.1 Beispiele . . . 21
2.1.2 Bandinschrift . . . 23
2.1.3 Konfiguration . . . 24
2.1.4 Beispiele . . . 25
2.2 Turing-Maschinen als Akzeptoren . . . 27
2.2.1 Initial- und Finalkonfiguration . . . 27
2.2.2 Sprache einer Turing-Maschine . . . 29
2.2.3 Beispiel einer Turing-Maschine für $a^n b^n c^n$ . . . 31
2.2.4 $\mathcal{L}_{ndetKA} \subset \mathcal{L}_{TM}$ . . . 33
2.2.5 $\mathcal{L}_0 = \mathcal{L}_{TM}$ . . . 34
2.3 Zusammensetzen von Turingmaschinen . . . 35
2.3.1 Die Rechtsmaschine $\mathbf{r}$ . . . 36
2.3.2 Die Linksmaschine $\mathbf{l}$ . . . 36
2.3.3 Die Zeichenmaschine $\boldsymbol{a_j}$ . . . 36
2.4 Spezielle Turing-Maschinen . . . 42
2.4.1 Die große Rechtsmaschine R . . . 43
2.4.2 Die große Linksmaschine L . . . 43
2.4.3 Die Rechts-Suchmaschine RS . . . 43
2.4.4 Die Links-Suchmaschine LS . . . 44
2.4.5 Die Suchmaschine S . . . 44
2.4.6 Die Kopiermaschine K . . . 45
2.4.7 Die linke Translationsmaschine T . . . 46
2.4.8 Die Verschiebemaschine V . . . 47
2.5 Turing-Berechenbarkeit . . . 48
2.5.1 Fleißige Biber . . . 49
2.6 Church-Turing-These . . . 56
2.7 Turing-Aufzählbarkeit Turing-Entscheidbarkeit . . . 57

2.8 Simulation von Turing-Maschinen . . . 58
2.8.1 Die Universelle Turing-Maschine . . . 59
2.8.2 Eine Sprache, die von keiner Turing-Maschine akzeptiert wird . . . 62
2.8.3 Turing-aufzählbare Sprachen . . . 63
2.8.4 Das Halteproblem . . . 65
2.9 Turing-Maschinen mit linearer Bandbeschränkung . . . 68

3 Berechenbare Funktionen **71**
3.1 Gödelisierung . . . 72
3.1.1 Gödelisierung von Turing-Maschinen . . . 75
3.2 Primitiv-rekursive Funktionen . . . 76
3.2.1 Beispiele für primitiv rekursive Funktionen . . . 80
3.2.2 Eigenschaften primitiv rekursiver Funktionen . . . 81
3.3 Die Ackermann-Funktion . . . 83
3.3.1 Primitiv rekursive Prädikate . . . 85
3.4 μ-rekursive Funktionen . . . 88
3.4.1 Der eingeschränkte μ-Operator . . . 91
3.4.2 μ-rekursive Funktionen . . . 93

4 Komplexitätstheorie **99**
4.1 Die Groß-O-Notation . . . 100
4.2 Das Erfüllbarkeitsproblem . . . 103
4.3 Färbung von Graphen . . . 107
4.4 Komplexitätsklassen . . . 109
4.5 NP-Vollständigkeit . . . 110
4.6 Der Satz von Cook . . . 111
4.7 NP-vollständige Probleme . . . 114

Kapitel 1

Algorithmen, Berechenbarkeit, Entscheidbarkeit

Im ersten Kapitel werden wir uns einige grundlegende Begriffe ansehen, die Ihnen gewiß schon aus früheren Vorlesungen bekannt sind.

Zunächst geht es um den Begriff **Algorithmus** , der zunächst intuitiv definiert wird. Im zweiten Kapitel werden wir eine Möglichkeit kennenlernen, Algorithmen exakt zu beschreiben.

Wir werden sehen, daß die Tatsache, daß es zur Berechnung einer Funktion einen Algorithmus gibt, eher eine Ausnahme darstellt. Für gewisse Fragestellungen kann man sich die Arbeit sparen, ein Programm zu entwickeln - es kann einfach keines geben...

Eng mit dem Begriff Berechenbarkeit ist der Begriff **Entscheidbarkeit** verknüpft. Hier geht es darum, festzustellen, ob z.B. eine **Menge** von Sprachen bestimmte Eigenschaften erfüllt.

1.1 Algorithmen

Das Wort *Algorithmus* leitet sich vom Namen des mittelalterlichen persischen Mathematikers

Abu Jaf'ar Mohammed ibn Mûsâ al-Khowârizm

her; er schrieb um das Jahr 825 ein mathematisches Lehrbuch mit dem Titel

Kitab al-jabr w'al-muqabala [7].

Das wohl bekannteste Beispiel für einen Algorithmus ist der **Euklidische Algorithmus** zur Bestimmung des **größten gemeinsamen Teilers** zweier Zahlen:

```
1. Wähle a > 0, b >= 0, o.B.d.A a >= b
2. b = 0 ==> ggt = a
3. Berechne r = a MOD b
4. r = 0 ==> ggt = b, STOP
5. Setze a = b, b = r, gehe nach 3
```


Bei der Formulierung haben wir angenommen, daß bekannt ist, wie `a MOD b` berechnet wird!
Für diese Berechnung kann man ebenfalls einen Algorithmus angeben, der bei geeigneter Darstellung von `a` und `b` sehr einfach wird:

```
1. a = n, b = m, a >= b, a >= 0, b > 0
2. a = 0 ==> a MOD b = 0
3. Stelle a durch n, b durch m Striche dar
4. Nimm solange m Striche von a weg, bis weniger als m
   Striche übrig bleiben.
5. Der verbleibende Rest ist das Ergebnis a MOD b
```

1.2 Eigenschaften eines Algorithmus

- **Allgemeinheit**
  Ein Algorithmus soll eine ganze **Problemklasse** mit unendlich vielen Ausprägungen lösen können. Eine Sammlung von Regeln, die nur $42*13$ berechnen kann, ist kein Algorithmus.

- **Endlichkeit**
  Diese Forderung gilt gleich in zweierlei Hinsicht.

  Ein Algorithmus soll durch einen **endlichen** Text beschreibbar sein.

  Angewendet auf **erlaubte** Eingaben, muß der Algorithmus nach endlich vielen Schritten **terminieren** . Das dürfen zwar **sehr viele** Schritte sein, aber eben immer nur **endlich viele**.

  So kann es keinen Algorithmus geben, der die Zahl π **vollständig** berechnet, da in endlich vielen Schritten immer nur eine endliche Anzahl von Dezimalstellen dieser **nichtperiodischen**, unendlich langen Zahl berechnet werden kann.

- **Determiniertheit**
  Die Ausgabedaten müssen **eindeutig** von den Eingabedaten abhängen. Auch die mehrmalige Anwendung des Algorithmus auf die gleichen Eingabedaten muß immer die gleiche Ausgabe erzeugen.

  Die **Elementaroperationen** des Algorithmus müssen eindeutig interpretierbar sein, d.h. für alle Situationen muß festgelegt sein, was zu geschehen hat.

Definition: (Algorithmus im intuitiven Sinn [9])
Es sei P eine **Problemklasse** und $\mathcal{A}$ die Menge der konkreten **Problemausprägungen** , d.h. die Menge derjenigen Daten, die ein Problem beschreiben.
Dann verstehen wir unter einem **Algorithmus** A_P zu der Klasse P ein allgemeines, determiniertes Verfahren, das – auf richtige Anfangsdaten $a \in \mathcal{A}$ angewendet – nach endlich vielen Schritten hält und die **Lösung** des Problems liefert (d.h. ein Element der Lösungsmenge $\mathcal{B}$).
Wir schreiben: $A_P : \mathcal{A} \longrightarrow \mathcal{B}$
d.h. $\forall a \in \mathcal{A} : \exists b \in \mathcal{B} : A_P(a) = b.$
Die Problemausprägungen beschreiben wir als Wort über einem **Alphabet** . Dabei ist es relativ gleichgültig, wie dieses Alphabet aussieht; ein einelementiges Alphabet ist ausreichend, um eine natürliche Zahl n als Wort der Länge n oder $n + 1$ darzustellen.

Beispiel:
$A = \{I\},\ 3 = III,\ 7 = IIIIIII$

oder
$A = \{0, 1, 2\},\ 3 = 10_3, 7 = 21_3, 27 = 1000_3$

1.3 Aufzähl- und Entscheidungsverfahren

Definition: (Entscheidungs- Aufzählverfahren [9])
Einen Algorithmus $A_P : \mathcal{A} \longrightarrow \mathcal{B}$ nennen wir auch

- **Entscheidungsverfahren**, falls $\mathcal{B} = \{true, false\}$ oder auch allgemeiner falls $|\mathcal{B}| = 2$
- **Aufzählverfahren**, falls $\mathcal{A} = \mathcal{N}$

Im ersten Fall wird eine binäre Entscheidung getroffen (z.B. Primzahl oder nicht).

Im zweiten Fall wird eine Folge realisiert $A_P(1), A_P(2), ... : A_P$ zählt diese Menge auf.

1.4 Berechenbare Funktionen

Man kann nun Algorithmen als Funktionen betrachten, die Eingabedaten auf Ausgabedaten abbilden.
Umgekehrt stellt sich die Frage, ob zu einer gegebenen Funktion immer ein Algorithmus existiert, der diese Funktion realisiert:

Definition: (berechenbare Funktion [9])
Es sei E ein Alphabet und $M_1, M_2 \subseteq E^*$.
Eine Funktion $f : M_1 \longrightarrow M_2$ heißt **berechenbar**, falls es einen Algorithmus $A_f : M_1 \longrightarrow M_2$ gibt, mit $A_f(w) = f(w)\ \forall w \in M_1$.

Zu jedem Argument $w \in M_1$ läßt sich also $f(w)$ in endlich vielen Schritten berechnen.

Beispiel:
$E = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Wir interpretieren Worte aus $E^+ = E^* \setminus \{\varepsilon\}$ als natürliche Zahlen. Dann ist $E^+ = \mathcal{N}_0$ und für $M_1 = \mathcal{N}_0$ und $M_2 = \mathcal{N}$ ist

$f : \mathcal{N}_0 \longrightarrow \mathcal{N}$ mit $f(n) = n + 1$

eine berechenbare Funktion, da es trivialerweise für diese Funktion einen Algorithmus gibt.

Wir werden aber gleich sehen, daß keineswegs zu jeder Funktion ein Algorithmus existiert.

1.5 Abzählbarkeit

Definition: (abzählbare Menge, überabzählbare Menge [9])
Es sei M eine Menge. Wir nennen M **abzählbar**, wenn sich M **bijektiv** auf $\mathcal{N}$ oder eine Teilmenge davon abbilden läßt.
Anderenfalls heißt M **überabzählbar**.

Beispiel:

Jede **endliche** Menge ist abzählbar.

Ist A ein Alphabet (d.h. endlich!), so ist A^* abzählbar. Man betrachte ganz einfach die **lexikographische Ordnung** .

Satz:
Es sei E ein Alphabet. Dann gibt es **überabzählbar viele** Funktionen $f : E^* \longrightarrow E^+$, von denen nur **abzählbar** viele berechenbar sind.

Beweis:
Jeder Algorithmus ist ein Text endlicher Länge. Diese Texte kann man lexikographisch anordnen und bijektiv auf $\mathcal{N}_0$ abbilden. Folglich gibt es nur abzählbar viele Algorithmen.
Dann kann es aber auch nur abzählbar viele Funktionen geben, für die ein Algorithmus existiert.
Nun gibt es aber überabzählbar viele Funktionen. Man betrachte z.B. die Funktionen $f : \mathcal{N}_0 \longrightarrow \{0, 1\}$. Wenn es denn nur abzählbar viele davon geben würde, könnten wir sie in einer Tabelle anordnen:

	w_1	w_2	w_3	w_4	w_5
f_1	$f_1(w_1)$	$f_1(w_2)$	$f_1(w_3)$	$f_1(w_4)$	$f_1(w_5)$
f_2	$f_2(w_1)$	$f_2(w_2)$	$f_2(w_3)$	$f_2(w_4)$	$f_2(w_5)$
f_3	$f_3(w_1)$	$f_3(w_2)$	$f_3(w_3)$	$f_3(w_4)$	$f_3(w_5)$
f_4	$f_4(w_1)$	$f_4(w_2)$	$f_4(w_3)$	$f_4(w_4)$	$f_4(w_5)$
f_5	$f_5(w_1)$	$f_5(w_2)$	$f_5(w_3)$	$f_5(w_4)$	$f_5(w_5)$

Betrachten wir nun die Funktion

$$f = \begin{cases} 1 & : \quad f_i(w_i) = 0 \\ 0 & : \quad f_i(w_i) \neq 0 \end{cases}$$

Dann muß f einen Index z.B. j haben: $f = f_j$. Nach Definition gilt aber: $f(w_j) \neq f_j(w_j)$.
Damit haben wir einen Widerspruch, den wir nur auflösen können, wenn wir die Annahme fallen lassen, daß wir die Funktionen abzählen können.

Folglich gibt es überabzählbar viele Funktionen, aber nur abzählbar viele Algorithmen.

1.6 Entscheidbare Mengen

Definition: (entscheidbare Menge [9])
Es seinen $M_1 \subseteq M_2 \subseteq E^*$.
M_1 heißt **entscheidbar relativ** zu M_2, wenn es einen Algorithmus $A_{M_1 M_2} : M_2 \longrightarrow \{true, false\}$ gibt, mit dessen Hilfe man für jedes Element aus M_2 feststellen kann, ob es zu M_1 gehört oder nicht:
$\forall w \in M_2 : A_{M_1 M_2}(w) ='' w \in M_1''$.

Es sei $M \subseteq E^*$.
M heißt **absolut entscheidbar** oder **entscheidbar** , wenn M relativ zu E^* entscheidbar ist.

Entscheidbarkeit ist also eine Eigenschaft einer **Menge**, nicht eines einzelnen Objekts. Man spricht auch von **entscheidbaren Problemen**:

"Es ist entscheidbar, ob die Sprache einer beliebigen regulären Grammatik endlich ist." (Wie ging das noch??)

Die Menge der Primzahlen ist entscheidbar.

Es ist für eine beliebige **Typ-0-Grammatik** G **nicht** entscheidbar, ob ein gegebenes Wort $w \in L(G)$ gilt oder nicht.

1.7 Charakteristische Funktionen

Auch ein Entscheidungsverfahren repräsentiert in natürlicher Weise eine berechenbare Funktion. Für die spezielle Gestalt dieser Funktion vergeben wir den folgenden Namen:

Definition: (charakteristische Funktion [9])
Es seien $M_1 \subseteq M_2 \subseteq E^*$. Dann heißt die Funktion

$\chi_{M_1,M_2} :\longrightarrow \{true, false\}$
mit

$$\chi_{M_1,M_2}(w) ::= \begin{cases} true & : \text{ falls } w \in M_1 \\ false & : \text{ falls } w \in M_2 \setminus M_1 \end{cases}$$

die **charakteristische Funktion** von M_1 bzgl. M_2.

Offenbar gilt:

Die Menge M_1 ist entscheidbar relativ zu $M_2 \Leftrightarrow$
χ_{M_1,M_2} ist berechenbar.

Beispiel:
Die charakteristische Funktion der Menge der Primzahlen M ist
$\chi_{M,\mathcal{N}} : \mathcal{N} \longrightarrow \{true, false\}$
mit

$$\chi_{M,\mathcal{N}}(n) ::= \begin{cases} true & : \text{ falls n Primzahl} \\ false & : \text{ sonst} \end{cases}$$

1.8 Aufzählbare Mengen

Eine weitere wichtige Eigenschaft von Mengen ist die **Aufzählbarkeit** .

Definition: (aufzählbare Menge [9])
Eine Menge $M \subseteq E^*$ heißt **aufzählbar**, wenn es eine Funktion $f : \mathcal{N}_0 \longrightarrow M$ gibt, die surjektiv und berechenbar ist.

Dann wird M durch f aufgezählt: $M = \{f(0), f(1), f(2), ...\}$

Eine **Aufzählverfahren** ist ein Algorithmus,
der diese Funktion realisiert.
Dieses Verfahren bringt die Elemente von M in eine feste Reihenfolge.
Dabei können einige Elemente auch **mehrfach** aufgezählt werden.

Den Zusammenhang zwischen **endscheidbar , aufzählbar** und **abzählbar** zeigt der folgende

Satz:

1. M ist aufzählbar $\Rightarrow$ M ist abzählbar; die Umkehrung gilt i.a. nicht.
2. M ist entscheidbar $\Leftrightarrow$ M **und** $E \setminus M$ sind aufzählbar;
   insbesondere gilt: M ist entscheidbar $\Rightarrow$ M ist aufzählbar.

Beweis:

1. M aufzählbar $\Rightarrow \exists f : \mathcal{N}_0 \longrightarrow M$, f surjektiv und berechenbar.Dann läßt sich M auch auf eine (Teilmenge) von $\mathcal{N}$ abbilden.
   Also ist M abzählbar.

   Die Anzahl der abzählbaren Mengen über einem Alphabet ist überabzählbar, wie man sich leicht mit einem Diagonalisierungsverfahren überlegen kann.

   Es gibt aber nur abzählbar viele berechenbare Funktionen.

   Folglich gibt es abzählbare Mengen, für die keine Funktion existiert, die diese aufzählen.

2. ($\Rightarrow$) $M \subseteq E^*$ entscheidbar. $M \neq E^*$ und $M \neq \emptyset \Rightarrow \exists w, w' \in E^*$ und $w \in M$ und $w' \notin M$. $w_0, w_1, w_2, ...$ sei eine Aufzählung von E^*, z.B. die lexikographische Ordnung.

   Da M entscheidbar ist, ist χ_{M,E^*} berechenbar und wir definieren für alle $i \in \mathcal{N}_0$

   $$f(i) ::= \begin{cases} w_i & : \text{ falls } \chi_{M,E^*}(w_i) = true \\ w & : \text{ sonst} \end{cases}$$

   $$f'(i) ::= \begin{cases} w_i & : \text{ falls } \chi_{M,E^*}(w_i) = false \\ w' & : \text{ sonst} \end{cases}$$

   Offenbar sind $f : \mathcal{N}_0 \longrightarrow M$ und $f' : \mathcal{N}_0 \longrightarrow E^* \setminus M$ surjektiv und berechenbar. Damit sind aber M und $E^* \setminus M$ aufzählbar.

   ($\Leftarrow$)
   Es gebe nun berechenbare Funktionen f und f', die M und $E^* \setminus M$ aufzählen.

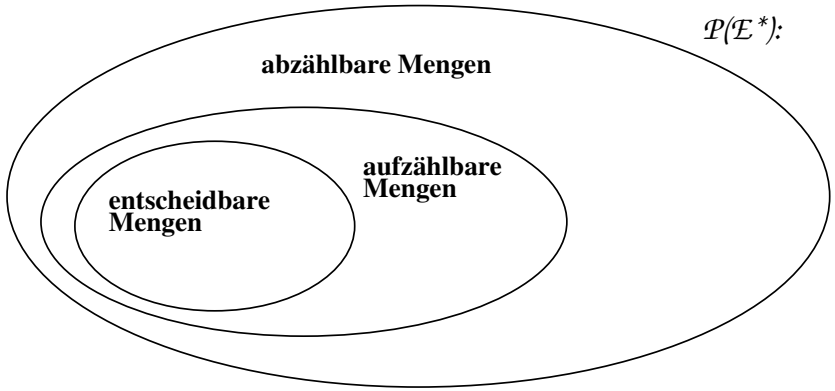
   Wir geben einen Algorithmus an, der M entscheidet:

   nacheinander werden $f(0), f'(0), f(1), f'(1), f(2), f'(2), ...$ berechnet. Wenn wir diesen Algorithmus auf ein Wort $w \in E^*$ anwenden, so wird dieses Wort irgendwann von f oder f' berechnet.

   Wird es von f berechnet liefern wir $true$,

   wird es von f' berechnet liefern wir $false$ ab.

Das folgende Bild zeigt noch einmal den Zusammenhang:



Kapitel 2

Turing-Maschinen

In diesem Kapitel wird das Konzept der **Turing-Maschine** vorgestellt. Eine solche Maschine ist eine **exakte** Präzisierung des Begriffs **Algorithmus** , den wir bisher eher intuitiv verwendet haben.

Jede **berechenbare Funktion** kann durch eine Turing-Maschine implementiert werden. Umgekehrt stellt jede Turing-Maschine eine berechenbare Funktion dar.

Eine Turing-Maschine ist wesentlich leistungsfähiger als ein Kellerautomat , da sie nicht der Einschränkung unterliegt, immer nur das oberste Symbol lesen oder manipulieren zu können.

Turing-Maschinen wurden von **Alan Turing** 1936 eingeführt – also **vor** Beginn des Computerzeitalters. Turing war im 2. Weltkrieg maßgeblich an der Entschlüsselung des **Enigma-Codes** der Wehrmacht beteiligt – er war somit keineswegs ein weltfremder Theoretiker. Seine interessante Biographie findet man in [4]. Im WWW sollte man unter `www.wadham.ox.ac.uk/~aho` nachsehen.

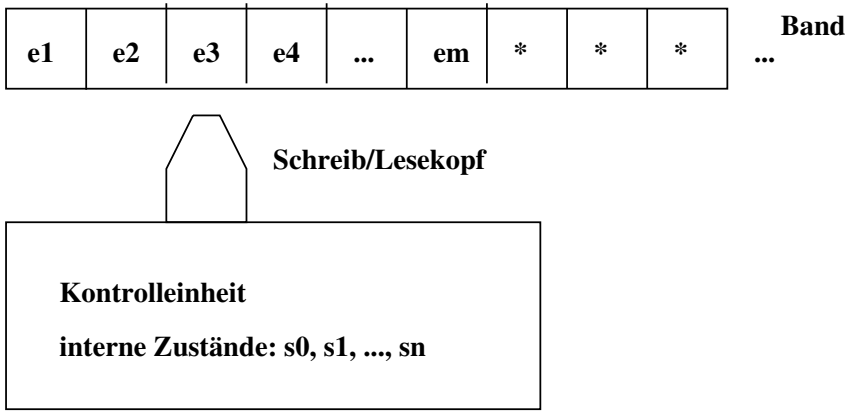
2.1 Das Maschinenmodell

Eine Turing-Maschine kann man sich als ein Gerät vorstellen, das aus einer **Kontrolleinheit** , einem **Schreib/Lesekopf** und einem einseitig unbegrenzten **Band** besteht.

Wie ein **endlicher Automat** kann die Kontrolleinheit endlich viele **interne Zustände** annehmen.

Das Band ist in einzelne **Felder** unterteilt, die genau ein Zeichen aufnehmen können.

Der Schreib/Lesekopf kann zu jedem Zeitpunkt genau ein Feld lesen oder verändern. Außerdem kann der Kopf (bzw. das Band) schrittweise nach links oder rechts bewegt werden.



Das **Eingabealphabet** enthält Symbole, die "irgendwie von außen" auf das Band geschrieben wurden.

Das **Bandalphabet** besteht aus allen Zeichen, die die Turing-Maschine verarbeiten kann. Das sind mindestens die Zeichen des Eingabealphabets und ein Spezialsymbol ($*$), das anzeigt, daß ein Feld leer/undefiniert ist.

Definition: (Turing-Maschine [9])
Eine **Turing-Maschine** T ist durch ein Sechstupel $T = (E, B, S, \delta, s_0, F)$ definiert, wobei
$E = \{e_1, ..., e_r\}$ das **Eingabealphabet** $(* \notin E)$,
$B = \{b_1, ..., b_m\}$ das **Bandalphabet** $(E \cup \{*\} \subseteq B)$,
$S = \{s_0, ..., s_n\}$ die **Zustandsmenge** ,
$\delta : S \times B \longrightarrow S \times B \times \{L, R, N\}$
eine **partielle** Funktion, die **Überführungsfunktion** ,
$s_0 \in S$ der **Anfangszustand** und
$F \subseteq S$ die **Menge der Endzustände** ist.

Die **Funktionsweise** einer Turing-Maschine ist durch die **Überführungsfunktion** festgelegt:

$(s, b) \mapsto (s', b', X)$

bedeutet, daß die Turing-Maschine , wenn im Zustand s das Zeichen b gelesen wird, in den Zustand s' wechselt und b durch b' ersetzt. Danach bewegt sich der Kopf um einen Schritt nach links ($X = L$), nach rechts ($X = R$) oder bleibt unverändert ($X = N$).

Üblicherweise stellt man δ als Tabelle (**Turing-Tafel**) dar:

	b_1	b	...	b_m
s_0				
s_1				
...				
s		(s', b', X)		
...				
s_n				

Die Turing-Maschine hält, falls δ für das aktuelle Paar (s, b) nicht definiert ist.

2.1.1 Beispiele

Turing-Maschine , die das letzte Zeichen des Eingabewortes auf dem Band sucht:

	0	1	*
s_0	$(s_0, 0, R)$	$(s_0, 1, R)$	$(s_1, *, L)$
s_1	—	—	—

Turing-Maschine , die das Eingabewort auf dem Band löscht:

	0	1	*
s_0	$(s_0, *, R)$	$(s_0, *, R)$	$(s_1, *, N)$
s_1	—	—	—

Turing-Maschine , die eine Zahl in **unärer** Darstellung um 1 vergrößert:

	1	*
s_0	$(s_0, 1, R)$	$(s_1, 1, R)$
s_1	—	—

Hier wird vorausgesetzt, daß die Turing-Maschine auf der ersten 1 steht, wenn sie gestartet wird.

Die folgende Turing-Maschine wird **links** von der Bandinschrift gestartet, sucht dann den Anfang der Eingabe, geht zum Ende und verlängert die Inschrift um 1.

	1	$*$
s_0	$(s_1, 1, R)$	$(s_0, *, R)$
s_1	$(s_1, 1, R)$	$(s_2, 1, R)$
s_2	—	—

2.1.2 Bandinschrift

Definition:(Bandinschrift)
Die **Bandinschrift** einer Turing-Maschine ist ein Wort über dem Bandalphabet, das mit dem Zeichen im ersten Feld des Bandes beginnt und endet

- mit dem Zeichen unter dem Schreib/Lesekopf, falls rechts vom Kopf nur Leerfelder stehen
- mit dem am weitesten rechts stehenden Feld, das kein Leerfeld ist, sonst

2.1.3 Konfiguration

Definition:(Konfiguration, Übergangsrelation [9])
Es sei $T = (E, B, S, \delta, s_0, F)$ eine Turing-Maschine .

Eine **Konfiguration** von T ist ein Tupel $(v, b, w, s) \in B^* \times B \times B^* \times S$. Dabei ist vbw als die aktuelle Bandinschrift und s als der aktuelle Zustand zu interpretieren. Ferner befindet sich der Schreib/Lesekopf auf dem Feld mit dem Zeichen b.

Die **Übergangsrelation** $\overset{\vdash}{}_T \subseteq (B^* \times B \times B^* \times S)^2$ beschreibt alle möglichen durch δ hervorrufbaren Übergänge einer Konfiguration zu einer Folgekonfiguration., d.h. für $v, v', w, w' \in B^*, s, s' \in S$ und $\delta(s, b) = (s', b', X)$ wird definiert:

(1) falls $X = N$:
$(v, b, w, s) \overset{\vdash}{}_T (v, b', w, s')$

(2) falls $X = R$:
$$(v, b, w, s) \overset{\vdash}{}_T \begin{cases} (vb', a, w', s') & : \quad w = aw' \\ (vb', *, \varepsilon, s') & : \quad w = \varepsilon \end{cases}$$

(3) Falls $X = L$ und $v = v'a \neq \varepsilon$
$$(v, b, w, s) \overset{\vdash}{}_T \begin{cases} (v', a, b'w, s') & : \quad w \neq \varepsilon \text{ oder } w = \varepsilon \text{ und } b' \neq * \\ (v', a, \varepsilon, s') & : \quad w = \varepsilon \text{ und } b' = * \end{cases}$$

$\overset{*}{\vdash}_T$ bezeichne wieder die reflexiv-transitive Hülle von $\overset{\vdash}{}_T$.

2.1.4 Beispiele

Turing-Maschine , die eine natürliche Zahl n, die durch n Striche auf dem Eingabeband dargestellt wird, verdoppelt:

	1	$*$
s_0	$(s_1, *, R)$	$(s_0, *, R)$
s_1	$(s_1, 1, R)$	$(s_2, *, R)$
s_2	$(s_2, 1, R)$	$(s_3, 1, L)$
s_3	$(s_3, 1, L)$	$(s_4, *, L)$
s_4	$(s_4, 1, L)$	$(s_5, 1, R)$
s_5	$(s_1, *, R)$	$(s_6, 1, R)$
s_6	$(s_6, 1, R)$	$(s_7, *, L)$
s_7	$(s_8, *, L)$	$(s_8, *, N)$
s_8	—	—

Was tut die folgende Turing-Maschine ?

	1	$*$
s_0	$(s_1, *, R)$	$(s_0, *, R)$
s_1	$(s_1, 1, R)$	$(s_2, 1, L)$
s_2	$(s_4, *, R)$	$(s_3, *, R)$
s_3	$(s_3, 1, R)$	$(s_6, 1, N)$
s_4	$(s_4, 1, R)$	$(s_5, 1, L)$
s_5	$(s_5, 1, L)$	$(s_2, 1, L)$
s_6	—	—

Turing-Maschine , die eine **Binärzahl**, die mindestens eine Null enthält, um 1 erhöht:

$T = (\{0, 1\}, \{0, 1, *\}, \{s_0, s_1, s_2\}, \delta, s_0, \{s_2\})$

	1	0	$*$
s_0	$(s_0, 1, R)$	$(s_0, 0, R)$	$(s_1, *, L)$
s_1	$(s_1, 0, L)$	$(s_2, 1, L)$	—
s_2	—	—	

2.2 Turing-Maschinen als Akzeptoren

In diesem Abschnitt wenden wir uns wieder dem Problem der Spracherkennung zu und betrachten eine Turing-Maschine wie schon früher einen endlichen Automaten oder einen Kellerautomaten als Akzeptor.

Dazu zunächst einige neue Begriffe:

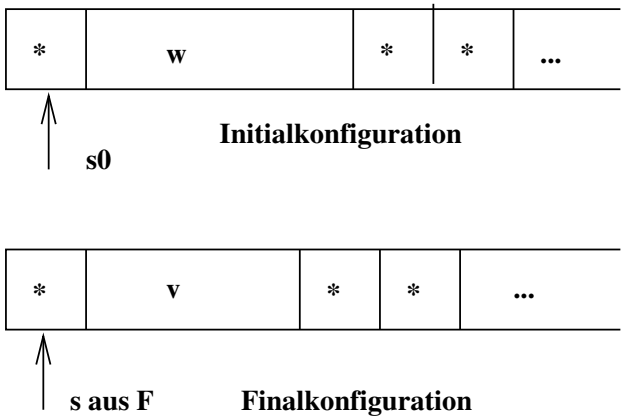
2.2.1 Initial- und Finalkonfiguration

Definition:(Initial- und Finalkonfiguration [9])
Es sei $T = (E, B, S, \delta, s_0, F)$ eine Turing-Maschine und $w \in E^*$ ein Eingabewort.
Dann nennen wir die Konfiguration $(\varepsilon, *, w, s_0)$ die **Initialkonfiguration** bzgl. w.

Die Konfiguration $(\varepsilon, *, v, s)$ mit $v \in B^*$ heißt **Finalkonfiguration** , falls es für $(\varepsilon, *, v, s)$ keine Folgekonfiguration gibt und $s \in F$ gilt.

In der Initialkonfiguration befindet sich der Schreib/Lesekopf unmittelbar links vom Wort w.

In der Finalkonfiguration ist kein weiterer Übergang möglich, die Turing-Maschine befindet sich in einem Endzustand, der Schreib/Lesekopf steht auf dem ersten Feld des Bandes und liest ein $*$.



2.2.2 Sprache einer Turing-Maschine

Definition:
Es sei $T = (E, B, S, \delta, s_0, F)$ eine Turing-Maschine .
T **akzeptiert** ein Wort $w \in E^*$, falls T beginnend mit der Initialkonfiguration bzgl. w nach endlich vielen Schritten in einer Finalkonfiguration anhält, d.h.
$(\varepsilon, *, w, s_0) \overset{*}{\underset{T}{\vdash}} (\varepsilon, *, v, s)$

wobei $s \in F$ und wobei es für $(\varepsilon, *, v, s)$ keine Folgekonfiguration gibt.
Die **Sprache** $L(T)$ einer Turing-Maschine ist die Menge aller Wörter, die T akzeptiert:
$L(T) = \{w \in E^* | (\varepsilon, *, w, s_0) \overset{*}{\underset{T}{\vdash}} (\varepsilon, *, v, s)$ ist eine Finalkonfiguration$\}$

Hier ist **nicht** gefordert, daß w **vollständig** gelesen wird.

Als Beispiel dafür, daß eine Turing-Maschine einen **DEA** simulieren kann, betrachten wir
$\mathcal{L} = \{w \in \{a,b\}^* | w$ enthält mindestens 2 a's $\}$
und eine Turing-Maschine , die $\mathcal{L}$ akzeptiert.

$T = (\{a,b\}, \{a,b,*\}, \{s_0,s_1,s_2,s_3\}, \delta, s_0, \{s_3\})$

	a	b	$*$
s_0	—	—	$(s_1,*,R)$
s_1	(s_2,a,R)	(s_1,b,R)	—
s_2	(s_3,a,L)	(s_2,b,R)	—
s_3	(s_3,a,L)	(s_3,b,L)	—

Bei der Analyse von *abbabb* wird nach dem Lesen des zweiten a der Endzustand erreicht.

2.2.3 Beispiel einer Turing-Maschine für $a^nb^nc^n$

Als Beispiel für die Mächtigkeit der Turing-Maschinen wollen wir jetzt eine Turing-Maschine betrachten, die die Sprache $a^nb^nc^n$ erkennt. Zunächst die Turing-Tafel:

	a	b	c	0	1	2	$*$
s_0							$(s_1,*,R)$
s_1	$(s_2,0,R)$				$(s_5,1,R)$		
s_2	(s_2,a,R)	$(s_3,1,R)$			$(s_2,1,R)$		
s_3		(s_3,b,R)	$(s_4,2,L)$			$(s_3,2,R)$	
s_4	(s_4,a,L)	(s_4,b,L)		$(s_1,0,R)$	$(s_4,1,L)$	$(s_4,2,L)$	
s_5					$(s_5,1,R)$	$(s_5,2,R)$	$(s_6,*,L)$
s_6				$(s_6,0,L)$	$(s_6,1,L)$	$(s_6,2,L)$	

Beim Start steht die Maschine unmittelbar links von der Bandinschrift.

$\boldsymbol{s_0}$ setzt den Kopf auf das erste Zeichen

$\boldsymbol{s_1}$ markiert ein $\boldsymbol{a}$ mit 0 und wechselt in $\boldsymbol{s_2}$, oder liest eine 1 und wechselt in $\boldsymbol{s_5}$, wenn kein $\boldsymbol{a}$ mehr vorhanden ist

$\boldsymbol{s_2}$ bewegt den Kopf nach rechts, um ein $\boldsymbol{b}$ zu finden und durch 1 zu markieren, und wechselt dann nach $\boldsymbol{s_3}$

$\boldsymbol{s_3}$ überliest $\boldsymbol{b}$s und markierte $\boldsymbol{c}$s, markiert dann das erste $\boldsymbol{c}$ mit 2 und wechselt in $\boldsymbol{s_4}$

$\boldsymbol{s_4}$ sucht die erste 0 von rechts, bleibt rechts davon stehen und wechselt in $\boldsymbol{s_1}$

$\boldsymbol{s_5}$ prüft, ob nur noch markierte Zeichen auf dem Band stehen

$\boldsymbol{s_6}$ fährt den Kopf an den Anfang

Aufgabe:
Geben Sie eine Turing-Maschine für $a^n b^n$ an!

2.2.4 $\mathcal{L}_{ndetKA} \subset \mathcal{L}_{TM}$

Satz:
Sei $\mathcal{L}_{TM}$ die Menge aller Sprachen, die durch Turing-Maschinen erkannt werden können: $\mathcal{L}_{TM} ::= \{L(T) | T \text{ Turing-Maschine }\}$.
$\mathcal{L}_{ndetKA}$ bezeichne die Menge aller Sprachen, die durch nichtdeterministische Kellerautomaten erkennbar sind. Dann gilt:
$\mathcal{L}_{ndetKA} \subset \mathcal{L}_{TM}$.

Beweis:
1. $\mathcal{L}_{ndetKA} \subset \mathcal{L}_{TM}$, weil jeder Kellerautomat durch eine Turing-Maschine simuliert werden kann. (Wie kann man sich das überlegen?)
2. $\mathcal{L}_{ndetKA} \neq \mathcal{L}_{TM}$, siehe Beispiel für $a^n b^n c^n$.

2.2.5 $\mathcal{L}_0 = \mathcal{L}_{TM}$

Wir können also mit Turing-Maschinen alle kontextfreien Sprachen und darüber hinaus noch einige Sprachen mehr erkennen. Mit einem erheblichen technischen Aufwand können wir zeigen, daß der folgende Satz gilt:

Satz:
Zu jeder Typ-0-Grammatik G existiert eine Turing-Maschine T mit $L(T) = L(G)$ und umgekehrt. Also gilt: $\mathcal{L}_0 = \mathcal{L}_{TM}$

Den **Beweis** findet man in [8].

2.3 Zusammensetzen von Turingmaschinen

Die Konstruktion komplexer Turing-Maschinen "from scratch" ist eine sehr mühsame Angelegenheit. Daher wollen wir uns nun auch ansehen, wie man ausgehend von einigen, wenigen Grundmodellen immer kompliziertere Turing-Maschinen zusammensetzen kann.

Dazu benötigen wir eben diese Grundmodelle und eine Möglichkeit das Zusammensetzen darzustellen. Die **Grundidee** beim Zusammenbau einer Turing-Maschine M aus den Mascinen M_1 und M_2 ist natürlich, daß M im Startzustand von M_1 gestartet, in den Startzustand von M_2 wechselt, sobald M_1 in einen Endzustand übergeht.

Bei den folgenden Betrachtungen gehen wir von Turing-Maschinen aus, die ein **beidseitig** unendliches Band besitzen. Das ändert nichts an der Funktionalität, macht aber einige Überlegungen einfacher.

Wir betrachten zunächst drei **Grundmodelle**:

1. *die kleine Linksmaschine* $\mathbf{l}$
2. *die kleine Rechtsmaschine* $\mathbf{r}$ und
3. *die Zeichenmaschinen* $\boldsymbol{a_j}$, die nur das Zeichen $a_j \in E$ auf das Band schreiben.

2.3.1 Die Rechtsmaschine r

	a_0	...	a_n
s_0	(s_1, a_0, R)	...	(s_1, a_n, R)
s_1	—	...	—

0	a_0	r	1
...	...	...	...
0	a_n	r	1
1	a_0	s	1
...	...	...	...
1	a_n	s	1

2.3.2 Die Linksmaschine l

	a_0	...	a_n
s_0	(s_1, a_0, L)	...	(s_1, a_n, L)
s_1	—	...	—

0	a_0	l	1
...	...	...	...
0	a_n	l	1
1	a_0	s	1
...	...	...	...
1	a_n	s	1

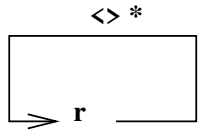
2.3.3 Die Zeichenmaschine a_j

	a_0	...	a_j	...	a_n
s_0	(s_0, a_j, N)	...	—	...	(s_0, a_j, N)

0	a_0	a_j	0
...	...	...	...
0	a_j	s	0
...	...	...	...
0	a_n	a_j	0

Eine Maschine, die ein Zeichen a_i auf das Band schreibt, einen Schritt nach rechts geht und dann stehen bleibt, kann man z.B. durch die Kombination von $\boldsymbol{a_i}$ und **r** erhalten. Wir drücken dies durch ein **Pfeildiagramm** aus:

Eine Maschine, die auf dem Band nach rechts läuft, bis ein Zeichen = $*$ gefunden wird, kann man darstellen als:



Allgemein ist ein solches Pfeildiagramm folgendermaßen aufgebaut:

1. Es existiert eine eindeutige **Startmaschine** . Sie kann durch einen Kreis gekennzeichnet werden, man erkennt sie aber auch daran, daß in sie kein Pfeil hinein führt.
2. Wenn zwei Maschinen durch einen Pfeil verbunden sind, an dem **keine** Bezeichnung steht, so sollen alle Zeilen der Turingtafel der ersten Maschine, in denen diese in einem Endzustand anhält, so geändert werden, daß die Maschine ohne Kopfbewegung in den Startzustand der zweiten Maschine übergeht.
3. Soll dieser Übergang nicht bei allen Eingabesymbolen erfolgen, so wird die **Bedingung** als Markierung des Pfeils angegeben.

Die Turingtafel der zusammengesetzten Maschine erhält man, in dem man zunächst alle Zustände der Teilmaschinen so umbenennt, daß keine Zustandsbezeichnung mehrfach auftritt. Wenn eine Maschine **mehrfach** vorkommt, muß die Turingtafel entsprechend oft aufgeführt werden!

Dann muß man in allen Zeilen der Matrix, in denen die Turing-Maschine in einem Endzustand anhält, den Übergang in den Startzustand der Folgemaschine eintragen, natürlich nur bei den Eingabesymbolen, die die Bedingung erfüllen.

Beispiel:

$\boldsymbol{a_i \rightarrow r \rightarrow r}$

	a_0	...	a_i	...	a_n
s_0	(s_0, a_i, N)	...	—	...	(s_0, a_i, N)

0	a_0	a_i	0
...	...	...	...
0	a_i	s	0
...	...	...	...
0	a_n	a_i	0

	a_0	...	a_n
s_0	(s_1, a_0, R)	...	(s_1, a_n, R)
s_1	—	...	—

0	a_0	r	1
...	...	...	...
0	a_n	r	1
1	a_0	s	1
...	...	...	...
1	a_n	s	1

	a_0	...	a_n
s_0	(s_1, a_0, R)	...	(s_1, a_n, R)
s_1	—	...	—

0	a_0	r	1
...	...	...	...
0	a_n	r	1
1	a_0	s	1
...	...	...	...
1	a_n	s	1

Umbenennen:

	a_0	...	a_i	...	a_n
s_0	(s_0, a_i, N)	...	—	...	(s_0, a_i, N)

0	a_0	a_i	0
...	...	...	...
0	a_i	s	0
...	...	...	...
0	a_n	a_i	0

	a_0	...	a_n
s_1	(s_2, a_0, R)	...	(s_2, a_n, R)
s_2	—	...	—

1	a_0	r	2
...	...	...	...
1	a_n	r	2
2	a_0	s	2
...	...	...	...
2	a_n	s	2

	a_0	...	a_n
s_3	(s_4, a_0, R)	...	(s_4, a_n, R)
s_4	—	...	—

3	a_0	r	4
...	...	...	...
3	a_n	r	4
4	a_0	s	4
...	...	...	...
4	a_n	s	4

Zusammensetzen:

	a_0	...	a_i	...	a_n
s_0	(s_0, a_i, N)	...	(s_1, a_i, N)	...	(s_0, a_i, N)
s_1	(s_2, a_0, R)	...	(s_2, a_i, R)	...	(s_2, a_n, R)
s_2	(s_3, a_0, N)	...	(s_3, a_i, N)	...	(s_3, a_n, N)
s_3	(s_4, a_0, R)	...	(s_4, a_i, R)	...	(s_4, a_n, R)
s_4	—	...	—	...	—

0	a_0	a_i	0
...	...	...	...
0	a_i	a_i	1
...	...	...	...
0	a_n	a_i	0
1	a_0	r	2
...	...	...	...
1	a_n	r	2
2	a_0	a_0	3
...	...	...	...
2	a_n	a_n	3
3	a_0	r	4
...	...	...	...
3	a_n	r	4
4	a_0	s	4
...	...	...	...
4	a_n	s	4

2.4 Spezielle Turing-Maschinen

Wir betrachten nun einige spezielle Turing-Maschinen , die sich aus den Maschinen $\boldsymbol{l, r, a_0, ..., a_n}$ zusammensetzen lassen. Dazu verwenden wir die Notation aus [3].

a_0	das uneigentliche Symbol $(*)$
m	ein Feld, das ein Symbol aus E enthält
$*$	ein leeres Feld
$\tilde{}$	ein Feld, das leer sein kann
$*...*$	eine endliche Folge leerer Felder (mind. eins)
$*...$	ein rechts leeres Bandende
W	ein Bandstück, das das nichtleere Wort W enthält. W enthält kein leeres Feld.
X	steht für $W_1 * W_2 * W_3 * ... * W_n$

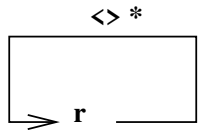
Das jeweilige Arbeitsfeld wird unterstrichen. Ist W unterstrichen ($\underline{W}$), so steht der Kopf auf dem letzten Zeichen von W.

2.4.1 Die große Rechtsmaschine R

Wirkungsweise:

$\underline{\tilde{}W} * \Rightarrow \tilde{}W\ \underline{*}$

$\underline{\tilde{}} * \Rightarrow \tilde{}\underline{*}$

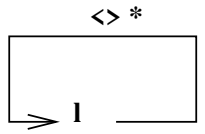


2.4.2 Die große Linksmaschine L

Wirkungsweise:

$*W\ \underline{\tilde{}} \Rightarrow \underline{*}W\ \tilde{}$

$*\ \underline{\tilde{}} \Rightarrow \underline{*}\tilde{}$

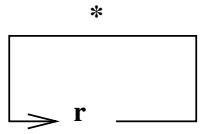


2.4.3 Die Rechts-Suchmaschine RS

Wirkungsweise:

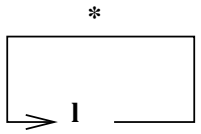
$\underline{\tilde{}} * ... * m \Rightarrow \tilde{}*...*\ \underline{m}$

$\underline{\tilde{}} m \Rightarrow \tilde{}\underline{m}$



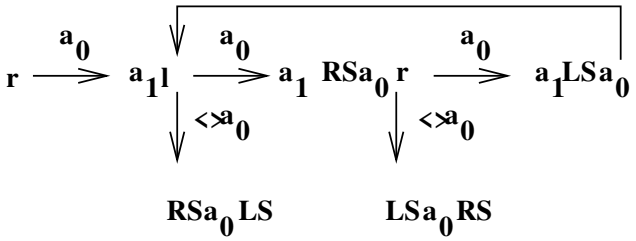
2.4.4 Die Links-Suchmaschine LS

Wirkungsweise:
$m * ...* \tilde{_}\Rightarrow \underline{m}*...* \tilde{}$
$m \tilde{_}\Rightarrow \underline{m}\tilde{}$



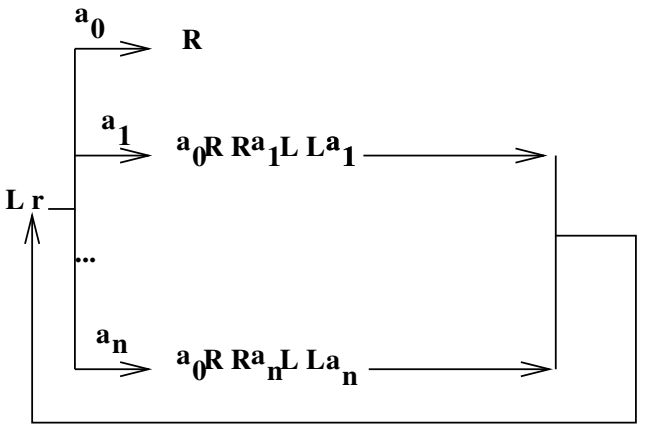
2.4.5 Die Suchmaschine S

Wirkungsweise:
sucht ein markiertes (nichtleeres) Feld



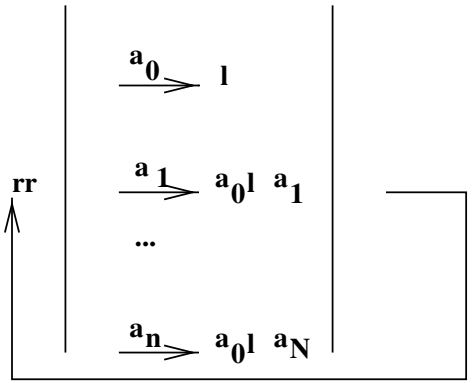
2.4.6 Die Kopiermaschine K

Wirkungsweise:
$*W\ \underline{*}... \Rightarrow *W * W\ \underline{*}...$
$*\ \underline{W}*... \Rightarrow *W * W\ \underline{*}...$



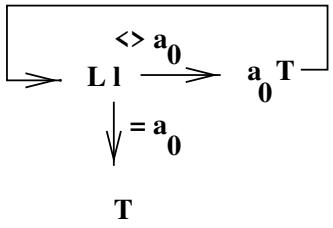
2.4.7 Die linke Translationsmaschine T

Wirkungsweise:
$\underline{\tilde{}}*W*\ldots \Rightarrow \tilde{}W\ \underline{*}\ldots$



2.4.8 Die Verschiebemaschine V

Wirkungsweise:
$*W_1*W_2\ \underline{*}\ldots \Rightarrow *W_2\ \underline{*}\ldots$



2.5 Turing-Berechenbarkeit

Den Begriff **berechenbar** haben wir bereits kennengelernt. Eine Funktion f war berechenbar, wenn ein Algorithmus existierte, der für jedes Argument x von f $f(x)$ berechnen konnte.

Jede Turing-Maschine implementiert einen Algorithmus. In Kürze werden wir uns überlegen, daß auch umgekehrt jeder Algorithmus durch eine Turing-Maschine beschrieben werden kann. Wir formulieren die Berechenbarkeit nun für Turing-Maschinen .

Definition: (Turing-Berechenbarkeit)
Eine Funktion $f(x_1, ..., x_n)$ heißt Turing-berechenbar, wenn eine Turing-Maschine existiert, die angesetzt auf die Bandinschrift $*x_1 * ... * x_n*$ nach endlich vielen Schritten auf dem ersten Zeichen von $f(x_1, ..., x_n)$ stehenbleibt.
Die x_i und $f(x_1, ..., x_n)$ sind dabei in unärer Darstellung.

Aufgabe:
Zeigen Sie, daß die Nachfolgerfunktion $SUCC(x)$ und die Addition $ADD(x, y)$ Turing-berechenbar sind!

Wir wissen bereits, daß es viele Funktionen gibt, die nicht berechenbar sind. Im nächsten Abschnitt werden wir eine weitere Funktion kennenlernen, die nicht berechenbar ist.
Wir verwenden die Begriff *berechenbar* und *Turing-berechenbar* synonym.

2.5.1 Fleißige Biber

Es ist simpel Turing-Maschinen anzugeben, die eine unendliche Zahl von Strichen auf ein anfangs leeres Band schreiben:

	|	*
s_0	—	$(s_1, \|, R)$
s_1	—	$(s_0, *, R)$

	|	*
s_0	—	$(s_0, \|, R)$

Man kann auch nicht-periodische Inschriften fabrizieren. Dazu schalten wir die Maschine, die einen einzigen Strich auf das leere Band schreibt und die Kopiermaschine in einer Endlosschleife hintereinander:

Interessanter sind Turing-Maschinen , die irgendwann einmal anhalten. 1982 dachte sich der ungarische Mathematiker **Tibor Rado** das **busy beaver Problem** aus:

Man nehme ein beidseitig unendliches Band, das Bandalphabet $\{|, *\}$ und eine Turing-Maschine mit n Zuständen $\{1, 2, ..., n\}$ und einem zusätzlichen Haltezustand H.
In jedem Schritt muß die Turing-Maschine ein Symbol schreiben und eine Bewegung nach links oder rechts machen oder aber anhalten. Die Maschine mit n Zuständen, die angesetzt auf das leere Band die meisten Striche schreibt, erhält den Titel **fleißiger Biber** . Die Strichfolge darf Lücken enthalten.

Beispiel:
Ein busy beaver mit 3 Zuständen

	$\|$	$*$
1	$(3, \|, L)$	$(2, \|, R)$
2	$(2, \|, R)$	$(1, \|, L)$
3	$(H, \|, R)$	$(2, \|, L)$
H	—	—

Nach 13 Schritten hat die Maschine 6 Striche auf das Band geschrieben und hält an.

Wir überlegen uns zunächst, daß für jedes n ein solcher fleißiger Biber existieren muß:
Die Zahl der Turingmaschinen mit n Zuständen und einem Bandalphabet mit zwei Elementen ist endlich. Also liste man alle Turing-Maschinen auf, und suche die Maschine heraus, die die meisten Striche produziert.

Wir können die Zahl der möglichen Turing-Maschinen sogar exakt angeben: n Zustände und ein zweielementiges Alphabet ergeben eine Turingtafel mit $2n$ Einträgen für die Nichtendzustände.
Jeder Eintrag besteht aus einem Tripel (Folgezustand, Symbol, Bewegung). Davon gibt es $(n + 1) * 2 * 2$. Macht zusammen $(4(n + 1))^{2n}$ mögliche Turingtafeln.

$\Sigma(n)$ bezeichne nun die maximale Anzahl von Strichen einer Turing-Maschine mit n Zuständen, $S(n)$ die maximale Anzahl der Schritte.
$\Sigma(n)$ ist eine wohldefinierte Funktion; für **kleine** Werte von n kann man die Funktionswerte auch ganz gut angeben.
Klein heißt hier, für Werte von 1 bis 4. Für größere Werte hat man nur einige **untere** Grenzen.

In [2] finden wir die folgende Tabelle:

n	$\Sigma(n)$
1	1
2	4
3	6
4	13
5	>= 4098

1984 wurde im **Scientific American** von einem 5-Zustands-Biber berichtet, der von dem deutschen Informatiker Uwe Schult gefunden wurde und 501 Striche produzierte. Der wurde später von einem anderen amerikanischen Biber abgelöst, der 1915 Striche produzierte. 1989 wurde dann in Deutschland wieder ein Biber gesichtet, der nach einer dreitägigen Jagd mit einem Hochgeschwindigkeitscomputer 4098 Striche lieferte.

Die folgende Turing-Maschine ist ein Kandidat für den Titel **busy beaver mit 6 Zuständen**:

	$\mid$	$*$
1	$(1, \mid, R)$	$(2, \mid, R)$
2	$(2, \mid, L)$	$(3, \mid, L)$
3	$(4, \mid, L)$	$(6, *, R)$
4	$(5, *, L)$	$(1, \mid, R)$
5	$(6, \mid, L)$	$(H, \mid, L)$
6	$(3, *, L)$	$(1, *, L)$
H	—	—

Sie sollten einmal überprüfen, daß dieser Biber 95.524.079 Striche produziert und nach 8.690.333.381.690.951 Schritten anhält. Das ist zwar eine ganze Menge, aber immer noch endlich...

Wir zeigen jetzt, daß weder $\Sigma(n)$ noch $S(n)$ berechenbar sind.

Die Idee ist, daß wir nachweisen, daß $\Sigma(n)$ **größer** als jede **beliebige** berechenbare Funktion ist, d.h. daß für jede berechenbare Funktion f gilt: $\exists n_0 \in \mathcal{N} : \ \forall n \geq n_0 : \ \Sigma(n) > f(n)$

Eine Turing-Maschine , die $f(n)$ berechnet, wird auf das Band angesetzt, auf dem n Striche stehen und hält nach endlich vielen Schritten mit der Bandinschrift von $f(n)$ Strichen.

Sei nun f eine beliebige berechenbare Funktion. Wir definieren

$F(x) = \Sigma_{i=0}^{x} f(i) + i^2$

Dann ist auch F berechenbar durch eine Turing-Maschine M_F. M_F habe n Zustände.

Wir betrachten eine Turing-Maschine M, die x Striche auf das zunächst leere Band schreibt und dann auf dem letzten Strich stehenbleibt. Das schafft sie mit x Zuständen.

Dahinter schalten wir M_F, die dann $F(x)$ Striche auf das Band schreibt. Das kostet noch einmal n Zustände. Wir setzten $M(F)$ ein weiteres Mal auf das Band an, auf dem nun $x + F(x)$ Striche stehen. So erhalten wir ein Band mit insgesamt $x + F(x) + F(F(x))$ Strichen. Diese Striche wurden von einer Maschine mit $x + n + n$ Zuständen geschrieben.

Eine fleißiger Biber mit $x + 2n$ Zuständen wird mindestens soviele Striche schreiben wie unsere Maschine. Also gilt:

$\Sigma(x + 2n) \geq x + F(x) + F(F(x))$.

Aber $F(x) = f(0) + 0^2 + f(1) + 1^2 + ... + f(x) + x^2 \geq x^2$

Außerdem: $\exists c_1 : \forall x \geq c_1 : x^2 > x + 2n$

Also $\forall x \geq c_1 : F(x) \geq x + 2n$

$F(x)$ ist aber monoton, d.h. $x > y \Rightarrow F(x) > F(y)$

folglich: $F(F(x)) > F(x + 2n)$.

Aber $\Sigma(x + 2n) \geq x + F(x) + F(F(x)) > F(F(x)) > F(x + 2n) =$

$f(0) + 0^2 + f(1) + 1^2 + ... + f(x + 2n) + (x + 2n)^2 > f(x + 2n)$

f war beliebig, berechenbar: folglich kann $\Sigma(n)$ nicht berechenbar sein.

Daß $S(n)$ nicht berechenbar ist, ist klar, da eine Turing-Maschine , die $\Sigma(n)$ Striche auf das Band schreibt, mindestens $\Sigma(n)$ Bewegungen machen muß. Also $S(n) >= \Sigma(n)$.

Der Beweis, den wir eben geführt haben, wurde bereits 1962 von **Rado** gefunden. 1964 entdeckte **M.W. Green** , daß die folgende Beziehung gilt:

f berechenbar $\Rightarrow f(\Sigma(n)) < \Sigma(n + 1)$

Wenn wir für f z.B. die Funktion

$f(m) = m^{m^{m^{\cdot^{\cdot^{\cdot^m}}}}}$

wählen. erhalten wir

$\Sigma(n + 1) > \Sigma(n)^{\Sigma(n)^{\Sigma(n)^{\cdot^{\cdot^{\cdot^{\Sigma(n)}}}}}}$

Nun kann man auch verstehen, daß es wenig aussichtsreich erscheint, nach fleißigen Bibern mit 6 und mehr Zuständen zu suchen.

2.6 Church-Turing-These

Wir haben gesehen, daß jede Turing-Maschine eindeutig einen Algorithmus repräsentiert. Um **beweisen** zu können, daß umgekehrt jeder Algorithmus einer Turing-Maschine entspricht, würden wir eine **exakte** Definition des Begriffs *Algorithmus* benötigen, die wir nur intuitiv gegeben haben.

Bisher ist es nicht gelungen, eine berechenbare Funktion anzugeben, eine Funktion also, für die ein Algorithmus existiert, die **nicht** auch Turing-berechenbar ist.

Daher gilt der folgende Satz, der auch als **Church-Turing-These** bekannt ist, heute als allgemein akzeptiert, obwohl man ihn wegen der **intuitiven** Definition des Begriffs *Algorithmus* nicht beweisen kann.

Satz: (Church-Turing-These)
Jede (im intuitiven Sinn) berechenbare Funktion ist auch Turing-berechenbar.

2.7 Turing-Aufzählbarkeit Turing-Entscheidbarkeit

Wir definieren nun die Begriffe **Aufzählbarkeit** und **Entscheidbarkeit** auch für Turing-Maschinen .

Definition:(Turing-Aufzählbarkeit, Turing-Entscheidbarkeit [9])

Sei A ein Alphabet, $M \subseteq A^*$. M heißt **Turing-entscheidbar** , falls die charakteristische Funktion χ_M Turing-berechenbar ist.

M heißt **Turing-aufzählbar** , wenn es eine Funktion $f : \mathcal{N} \longrightarrow M$ gibt, die surjektiv und Turing-berechenbar ist.

2.8 Simulation von Turing-Maschinen

In den vorangegangenen Abschnitten haben wir immer neue Turing-Maschinen für verschiedene Probleme erstellt. Wir werden nun eine Turing-Maschine betrachten, die sich ähnlich wie ein Computer **programmieren** läßt und dann beliebige Turing-Maschinen **simulieren** kann. Eine solche Maschine wird auch als **UTM** , als **Universelle Turing-Maschine** bezeichnet.

Turing-Maschinen sind sozusagen die Spitzenmodelle der Automaten, die wir zur Spracherkennung einsetzen können, da ja jeder nur denkbare Algorithmus durch eine Turing-Maschine ausgeführt werden kann. Dennoch gibt es Sprachen, die nicht von einer Turing-Maschine erkannt werden können, wie an sich leicht überlegt (wie eigentlich?).

Eine dieser Sprachen werden wir gleich kennenlernen.

Außerdem wollen wir eines der wichtigsten Probleme der Informatik, das sogenannte **Halteproblem** kennenlernen. Dieses Problem hat mit der **Nichtentscheidbarkeit** einer bestimmten Menge zu tun. (Wann war eine Menge eigentlich noch entscheidbar??)

Besorgen wir uns also zunächst einmal das notwendige Handwerkszeug.

2.8.1 Die Universelle Turing-Maschine

Die **UTM** soll so funktionieren, daß sie mit einer Bandinschrift, die die Codierung einer Turing-Maschine und die Eingabe für diese Turing-Maschine enthält, gestartet wird. Nun verhält sich die **UTM** so, wie die codierte Turing-Maschine , d.h. entweder hält sie nie an, weil die simulierte Maschine auch nicht stoppt, oder aber sie stoppt irgendwann mit der gleichen Bandinschrift.

Für die Codierung der Turing-Maschine gibt es verschiedene Möglichkeiten.

Wir betrachten eine **abzählbar unendliche** Menge S von Zuständen, von denen jede konkrete Turing-Maschine T eine endliche Teilmenge S_T verwendet.

$F \subseteq S$ ist eine abzählbar unendliche Menge von potentiellen **Endzuständen**.
$s_0 \in S$ ist immer der **Startzustand**.

Das **Bandalphabet** B bestehe immer aus $\{1, 0, *\}$.

Die Codierung der **Überführungsfunktion** einer Turing-Maschine T sieht nun so aus:

Jeder Übergang $(s_j, b_i) \mapsto (s_k, b_r, x_m),\ x_m \in \{L, R, N\}$ wird codiert als

$$0\underbrace{11...11}_{j}0\underbrace{11...11}_{i}0\underbrace{11...11}_{k}0\underbrace{11...11}_{r}0\underbrace{11...11}_{m}0$$

Die Codierungen aller möglichen Übergänge stehen hintereinander auf dem Band.

Die aktuelle Konfiguration von T muß ebenfalls codiert werden. Eine solche Konfiguration besteht aus einem Tupel (v, b, w, s) wobei v die Bandinschrift links vom S/L-Kopf, b das aktuelle Symbol, w die Inschrift rechts vom Kopf und s den Zustand bezeichnet. Alle diese Informationen können wie oben zeichenweise codiert werden.

Die Codierung der Überführungsfunktion legt sie zu simulierende Turing-Maschine eindeutig fest. Die verwendete Folge aus 0 und 1 kann man nun als (zugegeben möglicherweise sehr große **Binärzahl**) betrachten, die die Turing-Maschine eindeutig charakterisiert.

Definition: (Simulation einer Turing-Maschine [9])
Es bezeichne C_{TM} bzw. C_{KON} die oben beschriebene Codierung der Überführungsfunktion und der Konfiguration von Turing-Maschinen .
Eine universelle Turing-Maschine UTM **simuliert** eine Turing-Maschine T, wenn gilt:
U befindet sich zu Anfang in der Initialkonfiguration
$(\varepsilon, *, C_{TM}(T)C_{KON}(k_0), s_0)$, k_0 Initialkonfiguration von T
falls T die Konfigurationen $k_0 \vdash ... \vdash k_n \vdash ...$ durchläuft,
durchläuft UTM die Konfigurationen

$$(\varepsilon, *, C_{TM}(T)C_{KON}(k_0), s_0) \overset{*}{\underset{UTM}{\vdash}}$$
$$(\varepsilon, *, C_{TM}(T)C_{KON}(k_1), s_1') \overset{*}{\underset{UTM}{\vdash}}$$
$$... \overset{*}{\underset{UTM}{\vdash}}$$
$$(\varepsilon, *, C_{TM}(T)C_{KON}(k_n), s_n') \overset{*}{\underset{UTM}{\vdash}} ...$$

UTM hält (in einer Finalkonfiguration) genau dann,
wenn T (in einer Finalkonfiguration) hält.

Es gibt noch andere Verfahren Turing-Maschinen zu codieren, insbesondere kann man die Codierung wesentlich kompakter gestalten. Eine ausführliche Beschreibung einer effizienteren Codierung findet man z.B. in [7].

2.8.2 Eine Sprache, die von keiner Turing-Maschine akzeptiert wird

Da wir jeder Turing-Maschine eine Zahl zuordnen können, kann man von der 1., 2., ..., i-ten Turing-Maschine sprechen.

Satz:
Die Sprache $L_{NA} := \{w_i \mid w_i \notin L(T_i)\}$ wird von keiner Turing-Maschine akzeptiert.

Beweis:
Angenommen die Turing-Maschine T_j akzeptiert L_{NA}.

Dann gilt $L_{NA} = L(T_j)$.

Nun betrachten wir w_j.

$w_j \in L_{NA} \Leftrightarrow w_j \in L(T_j)$ (denn $L_{NA} = L(T_j)$)

$\Leftrightarrow w_j \notin L_{NA}$ (wegen Definition L_{NA})

Widerspruch!!

2.8.3 Turing-aufzählbare Sprachen

Die Turing-aufzählbaren Sprachen sind genau die Sprachen, die auch von Turing-Maschinen **akzeptiert** werden. Diese weitere Charakterisierung der Sprachklasse $\mathcal{L}_{TM}$ besagt der folgende

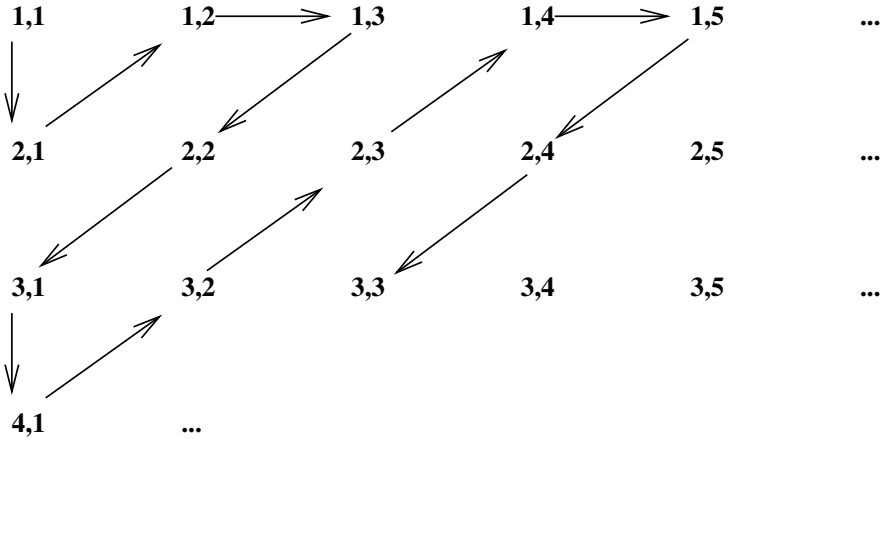
Satz:
Sei E ein Alphabet, $M \subseteq E^*$. Dann gilt:
M ist Turing-aufzählbar $\Leftrightarrow M \in \mathcal{L}_{TM}$.
Wenn $\mathcal{L}_A$ bzw. $\mathcal{L}_{TM-A}$ die Menge der aufzählbaren bzw. Turing-aufzählbaren Sprachen bezeichnen, gilt also:
$\mathcal{L}_A = \mathcal{L}_{TM-A} = \mathcal{L}_{TM} = \mathcal{L}_0$

Beweis:
"$\Rightarrow$": Sei M Turing-aufzählbar. Dann existiert eine surjektive Funktion $f : \mathcal{N}_0 \to M$, die Turing-berechenbar ist.
Sei T_f die Turing-Maschine , die f berechnet. Wir verwenden die UTM, um T_f mit den Eingaben $0, 1, 2, 3, ...$ zu simulieren.
Wir denken uns nun eine weitere Turing-Maschine T, die die Zahlen $0, 1, 2, 3, ...$ produziert und außerdem T_f simuliert.

T funktioniert nun so, daß sie, angesetzt auf ein Wort w, nacheinander die Zahlen $0, 1, 2, ...$ produziert und durch Simulation von T_f $f(0), f(1), f(2), ...$ berechnet. Ist irgendwann $f(i) = w$ hält T an.

"$\Leftarrow$": Sei $M \in \mathcal{L}_{TM}$, d.h. es gibt eine Turing-Maschine , die M akzeptiert. Gesucht ist eine Turing-Maschine T_f, die nacheinander die Funktionswerte $f(0), f(1), f(2), ... \in M$ produziert, und genau alle Elemente aus M liefert.

Wir konstruieren eine Turing-Maschine T, die alle Zahlenpaare (i, j) generiert, und zwar in der folgenden Reihenfolge:



Ferner soll T die Worte $w_0, w_1, w_2, ... \in E^*$ aufzählen.
T generiert nun die Zahlenpaare (i, j) in der besagten Reihenfolge. Für jedes Paar (i, j) wird dann für i Schritte die Turing-Maschine TM_M auf w_j losgelassen. Ist $w_j \in M$, so wird irgendwann ein Paar (i, j) erzeugt, so daß w_j innerhalb von i Schritten akzeptiert wird.
Zu jeder Eingabe $k \in \mathcal{N}_0$ werden nun die Zahlenpaare generiert und das (k+1)-te akzeptierte Wort ausgegeben.
Damit haben wir eine Turing-Maschine gefunden, die $\mathcal{N}_0$ surjektiv auf M abbildet.

2.8.4 Das Halteproblem

Schön wär's ja, wenn wir einen Compiler um die Option *"suche Endlosschleifen"* erweitern könnten. Dann könnte der Compiler beim Compilieren eine Warnung ausgeben, daß ein Programm möglicherweise nicht terminiert. Leider geht das nicht, wie der folgende Satz zeigt:

Satz:
Das Halteproblem für Turing-Maschine ist unentscheidbar, d.h. es gibt keine Turing-Maschine , die für jede **beliebige** Turing-Maschine T und jedes **beliebige** Eingabewort w entscheiden kann, ob T angesetzt auf w anhält oder nicht.

Beweis:
Wir werden diesen Satz gleich zweimal beweisen (weil's so schön ist!).

1. Angenommen, es gäbe eine solche Turing-Maschine T_e. Wir zeigen, daß es dann auch eine Turing-Maschine gibt, die die Sprache L_{NA} aus dem letzten Abschnitt akzeptiert. Und das kann ja nicht sein.

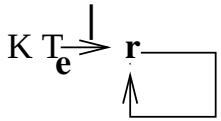
Wir skizzieren die Arbeitsweise dieser Turing-Maschine :

- Sei $w \in E^*$. Bestimme $i \in \mathcal{N}$ mit $w = w_i$.
- Bestimme T_i.
- Bestimme mit T_e, ob T_i angesetzt auf w_i hält.
- Wenn T_i nicht anhält, akzeptiere w_i.
- Wenn T_i anhält, akzeptiere w_i, falls T_i w_i **nicht** akzeptiert. Das kann man mit der UTM simulieren.

Damit ist L_{NA} sogar entscheidbar! Und das kann ja nun wirklich nicht wahr sein!

2. Man kann sich auch von der Nichtexistenz der Turing-Maschine T_e so überzeugen:

Wenn T_e existiert, dann bauen wir die folgende Turing-Maschine



Diese Maschine nennen wir A. T_e wenden wir auf eine Eingabe w und eine Codierung einer Turing-Maschine X an.

Wenn X bei der Eingabe w anhält, so soll T_e auf $|$, sonst auf $*$ stehenbleiben.

Wenn wir jetzt A auf die Codierung von X loslassen, wird erst einmal X kopiert. Dann prüft T_e, ob X angesetzt auf X hält oder nicht. Wenn X hält, so läuft A in eine Endlosschleife.

Für A existiert auch eine Codierung, die wir nun auf das Band schreiben.

Wenn A nun losläuft, erhalten wir einen Widerspruch, da A anhält, wenn A nicht anhält und umgekehrt.

2.9 Turing-Maschinen mit linearer Bandbeschränkung

Wir haben gesehen, daß Typ-2-Sprachen durch nichtdeterministische Kellerautomaten erkannt werden, Typ-0-Sprachen durch Turing-Maschinen . Zwischen diesen beiden Sprachklassen liegen die kontextsensitiven oder Typ-1-Sprachen.
Auch für diese Sprachen kann man eine Klasse von erkennenden Automaten angeben, die genau diese Sprachen akzeptieren und mit ihren Fähigkeiten zwischen den Kellerautomaten und Turing-Maschinen liegen.

Wir machen bei den bekannten Turing-Maschinen nur die Einschränkung, daß bei der Analyse eines Wortes nur ein **endlich langes Stück** des Bandes benutzt werden darf. Die Länge dieses Stücks ist **linear** abhängig von der Länge des Eingabewortes.
Ferner fordern wir, daß die Turing-Maschine **nichtdeterministisch** arbeiten soll.

Formaler heißt das:

Definition: (linear beschränkter Automat [9])
Ein **linear beschränkter Automat** ist eine nichtdeterministische Turing-Maschine $LBA = (E, B, S, \delta, s_0, F)$ mit E, B, s_0, F wie gehabt. Das Band sei rechtsseitig unendlich, $\$ \in B$ und δ eine **totale** Funktion:

$$\delta : S \times B \to \mathcal{P}(S \times B \times \{L, R, N\})$$

Auf dem Band steht das Begrenzungszeichen \$ und der Schreib/Lesekopf kann sich nur zwischen dem linken Bandende und dem \$ bewegen.
Die verfügbare Bandlänge ist abhängig vom Eingabewort auf $k * |w| + c$ Felder begrenzt.

Dann gilt der folgende

Satz:
Zu jeder Typ-1-Grammatik gibt es einen linear beschränkten Automaten LBA und umgekehrt mit $L(LBA) = L(G)$.
Insgesamt gilt also $\mathcal{L}_1 = \mathcal{L}_{LBA}$

Beispiel: Die Turing-Maschine , die $a^n b^n c^n$ erkennt, verwendet nur die Felder, auf denen das Eingabewort steht, bei der Analyse. Mit kleinen Änderungen (Begrenzungssymbol) kann man sie in einen LBA umbauen.

Bemerkung:
Es ist unbekannt, ob deterministische und nichtdeterministische LBA unterschiedliche Sprachfamilien erkennen.

Kapitel 3

Berechenbare Funktionen

Die Verwendung von Turing-Maschinen erlaubte eine exakte Charakterisierung des Berechenbarkeitsbegriffs. In der Praxis erweisen sich Turing-Maschinen aber als sehr unhandlich. Daher werden wir uns nun eine andere Möglichkeit ansehen, Berechenbarkeit zu charakterisieren.

Zur Vereinfachung betrachten wir nur Funktionen, die mit natürlichen Zahlen arbeiten. Daß dies keine Einschränkung darstellt, zeigt der folgende Abschnitt.

3.1 Gödelisierung

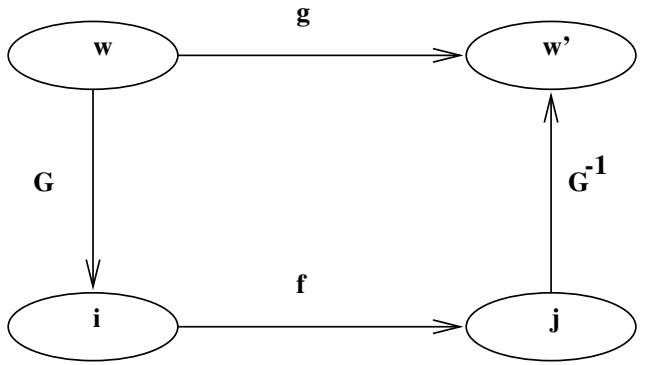
Der Begriff **Gödelisierung** geht auf den Mathematiker **Kurt Gödel** zurück, der in den 30er Jahren entdeckte, daß bestimmte formale Systeme, darunter auch die Arithmetik, erlauben, wahre Formeln zu formulieren, die man in dem betreffenden Kalkül nicht beweisen kann.
Die Idee bei der Gödelisierung ist die folgende:

E^* ist abzählbar. Daher kann man die Worte aus E^* in eine Reihenfolge bringen: $w_0, w_1, w_2, w_3, ...$
Nun betrachtet man eine Funktion $g : E^* \to E^*$ als eine Funktion $f : \mathcal{N}_0 \to \mathcal{N}_0$, wobei gilt:
$f(i) = j \Leftrightarrow g(w_i) = w_j$
Man muß aber sicherstellen, daß aus der Berechenbarkeit von f die von g folgt und umgekehrt. Daher stellen wir an die Reihenfolge der Worte einige Bedingungen:

Definition: (Gödelisierung [9])
Eine Abbildung
$G : E^* \to \mathcal{N}_0$
wird **Gödelisierung** von E^* genannt, wenn gilt

1. G ist injektiv,
2. G ist berechenbar
3. $G(E^*)$ ist entscheidbar
4. die Umkehrfunktion $G^{-1} : G(E^*) \to E^*$ ist berechenbar

Ist $G : E^* \to \mathcal{N}_0$ eine Gödelisierung von E^*,
so heißt $G(w)$ die **Gödelnummer** von w für alle $w \in E^*$.



Für jedes $w \in E^*$ kann man die Berechnung von $g(w)$ zurückführen auf:
$G(w) = i, f(i) = j, G^{-1} = w'$

Beispiel:
Betrachte $E = \{a, b\}$. Setze $a = 1, b = 2$.
Codiere $w \in E^*$ als Zeichenfolge über $\{1, 2\}$, z.B.
$aaa \mapsto 111, bab \mapsto 212$
Interpretiere die erhaltene Zahl als Zahl im Dreiersystem und konvertiere ins Dezimalsystem:
$(111)_3 \mapsto 13_{10}, (212)_3 \mapsto 23_{10}$
Man rechne nach, daß die Bedigungen der Definition erfüllt sind.

Satz: Sei G eine Gödelisierung von E^*, g eine Abbildung von $E^* \to E^*$.
Definiert man $f : G(E^*) \to G(E^*)$ durch
$f(i) = j \Leftrightarrow i = G(w), j = G(w'), g(w) = w'$,
dann ist f berechenbar genau dann, wenn g berechenbar ist.

3.1.1 Gödelisierung von Turing-Maschinen

Wir betrachten Turing-Maschinen mit n Zuständen $z_1, ..., z_n$.

Jeden Zustand codieren wir durch seinen Index: $g_Z(z_j) = j$.

Die Zeichen des Bandalphabets $\{e_1, ..., e_r\}$
codieren wir durch $g_E(e_i) = i$.

Die Bewegungen l, r, s stellen wir durch $r + 1, r + 2$ und $r + 3$ dar.

Die Turing-Tafel enthalte m Zeilen der Form $z_i e_j e_k z_l$.

Diese Tafel codieren wir als $m \times 4$-Matrix a_{ij},
mit Zeilen der Form $g_Z(z_i)g_E(e_j)g_E(e_k)g_Z(z_l)$.

Jetzt kommt der Trick:

wir betrachten die Folge der **Primzahlen** $p_1 = 2, p_2 = 3, p_3 = 5, ...$

und bilden die Zahl $p_1^n * p_2^r * p_3^{a11} * p_4^{a12} * ....$

Dann ist entscheidbar, ob eine Zahl n Gödelnummer einer Turing-Maschine M ist.

Dazu zerlegt man n in seine Primfaktoren, die Potenzen von 2 und 3 dürfen nicht 0 sein, und aus den restlichen Potenzen muß sich eine Turing-Tafel erstellen lassen.

3.2 Primitiv-rekursive Funktionen

In diesem Abschnitt werden wir die Struktur berechenbarer Funktionen dadurch analysieren, daß wir von einem festen Satz einfacher **Grundfunktionen** ausgehen, aus denen dann kompliziertere Funktionen zusammengesetzt werden.

Eine der Grundfunktionen ist die **Nachfolgerfunktion**

$S : \mathcal{N}_0 \to \mathcal{N}_0$,

mit der man die **Addition** beschreiben kann:

$add(m, n) ::= \underbrace{S(S(S(S...(n))...)}_{\text{m-mal}}.$

Oder

$add(n, 0) = n$
$add(n, S(m)) = S(add(n, m))$

Für die **Multiplikation** erhält man:

$mult(n, 0) = 0$

$mult(n, S(m)) = add(n, mult(n, m))$

Frage: Wie sieht die **Exponentiation** aus?

Wir haben zwei **Zusammensetzungsprozesse** benutzt:

1. **Einsetzung :**
   eine Funktion wird auf das Ergebnis einer anderen angewendet.
2. **Rekursion :**
   eine Funktion wird durch Anwendung auf sich selbst erklärt.

Nun definieren wir, welche **Basisfunktionen** wir zulassen wollen. Mit den Zusammensetzungsregeln erhalten wir so die Menge der **primitiv rekursiven Funktionen**.

Definition: (primitiv rekursive Funktionen [9])
Die Menge $\mathcal{P}$ der **primitiv rekursiven Funktionen** enthält genau die folgenden Funktionen:

1. **die Basisfunktionen**
   - die **Nachfolgerfunktion** $S : \mathcal{N}_0 \to \mathcal{N}_0$
     definiert als $S(n) ::= n + 1$. Statt $S(n)$ schreiben wir auch n'.
   - für alle $q, r \in \mathcal{N}_0$ die **konstante Funktion** $C_q^r : \mathcal{N}_0^r \to \mathcal{N}_0$, die für jedes Tupel $(n_1, ..., n_r)$ den Wert $C_q^r(n_1, ..., n_r) ::= q$ liefert.
   - für alle $i, r \in \mathcal{N}_0$ mit $1 \leq i \leq r$ die **Projektionsfunktion**
     $U_i^r : \mathcal{N}_0^r \to \mathcal{N}_0$,
     die für jedes Tupel $(n_1, ..., n_r)$ den Wert $U_i^r(n_1, ..., n_r) ::= n_i$ liefert.

2. Sind $f : \mathcal{N}_0^r \to \mathcal{N}_0, r \in \mathcal{N}_0$ und $g_1, ..., g_r : \mathcal{N}_0^m \to \mathcal{N}_0$ in $\mathcal{P}$, so ist auch die Funktion $h : \mathcal{N}_0^m \to \mathcal{N}_0$ mit

   $h(n_1, ..., n_m) ::= f(g_1(n_1, ..., n_m), ..., g_r(n_1, ..., n_m))$

   in $\mathcal{P}$.

3. Sind $f : \mathcal{N}_0^r \to \mathcal{N}_0, r \in \mathcal{N}_0$ und $g : \mathcal{N}_0^{r+2} \to \mathcal{N}_0$ in $\mathcal{P}$, so ist auch die Funktion $h : \mathcal{N}_0^{r+1} \to \mathcal{N}_0$ in $\mathcal{P}$, die den folgenden Rekursionsgleichungen genügt:

   1. $h(0, n_1, ..., n_r) ::= f(n_1, ..., n_r)$
   2. $h(S(n), n_1, ..., n_r) ::= g(n, h(n, n_1, ..., n_r), n_1, ...n_r)$.

   h entsteht durch **primitive Rekursion** aus f und g.

Eine Funktion ist genau dann in der Menge der primitiv rekursiven Funktionen enthalten, wenn sie entweder eine Basisfunktion ist, oder durch endlich viele Anwendungen der Einsetzung und/oder primitiven Rekursion erhalten werden kann.

Aufgabe: Geben Sie die Turing-Maschinen für die Berechnung der Basisfunktionen an!

Satz:
Jede primitiv rekursive Funktion $f : \mathcal{N}_0^r \to \mathcal{N}_0, r \geq 0$ ist eine wohldefinierte totale Funktion, d.h. sie ist für alle $(n_1, ..., n_r) \in \mathcal{N}_0^r$ definiert und eindeutig bestimmt.

Man sieht dies bei den Basisfunktionen leicht ein und muß dann durch Induktion über den Aufbau der Funktionen zeigen, daß diese Eigenschaften bei der Zusammensetzung erhalten bleiben.

Beispiel:
Die **Addition** ist eine primitiv rekursive Funktion , die man aus der Nachfolgerfunktion durch primitive Rekursion erhält:

$add(0, n_1) = U_1^1(n_1)$
$add(S(n), n_1) = S(U_2^3(n, add(n, n_1), n_1))$

Das kann man vereinfachen:

$add(0, n_1) = n_1$
$add(S(n), n_1) = S(add(n, n_1))$

Frage:
Wie definiert man die Multiplikation nach diesem Schema?

Offenbar kann man, wenn einige Argumente nicht benötigt werden, das Rekursionsschema in einer **vereinfachten Form** anwenden, ohne daß sich an der primitiven Rekursion etwas ändert.

3.2.1 Beispiele für primitiv rekursive Funktionen

Polynome mit natürlichzahligen Koeffizienten sind primitiv rekursiv :
$f(n) ::= an^2 + bn + c$ kann man schreiben als

$f(n) = add(add(mult(a, mult(n, n)), mult(b, n)), c)$

$\boldsymbol{x^y}$ ist primitiv rekursiv :
$f(y, x) = x^y$

$f(0, x) = C_0^1() = 1$
$f(y', x) = mult(x, f(y, x))$

die **Vorgängerfunktion** ist primitiv rekursiv :

$$f(n) ::= \begin{cases} 0 & : \quad n = 0, f(0) = 0 \\ n - 1 & : \quad n \geq 1, f(n') = n \end{cases}$$

3.2.2 Eigenschaften primitiv rekursiver Funktionen

Der Funktionswert einer primitiv rekursiven Funktion ist immer definiert und eindeutig bestimmt.
Bei Anwendung der primitiven Rekursion oder Einsetzung läßt er sich aus den Werten der Basisfunktionen algorithmisch berechnen. Man kann also feststellen:

Satz:
Jede primitiv rekursive Funktion ist Turing-berechenbar.

Den recht aufwendigen Beweis findet man in [3].

Die Umkehrung dieses Satzes gilt nicht, denn:

Turing-berechenbare Funktionen müssen im Gegensatz zu primitiv rekursiven Funktionen nicht total sein. Also kann es vorkommen, daß eine Turing-Maschine für bestimmte Werte nicht in einer Finalkonfiguration anhält.

Aber auch wenn man die totalen Turing-berechenbaren Funktionen betrachtet, stellt man fest, daß es Turing-berechenbare Funktionen geben muß, die nicht primitiv rekursiv sind.

Sei $f : \mathcal{N}_0 \to \mathcal{N}_0$ eine primitiv rekursive Funktion.
Die Menge der primitiv rekursiven Funktionen ist, wie **Péter** 1951 bewiesen hat, aufzählbar. Man kann die Funktionen also in eine Reihenfolge bringen und zu jedem $i \in \mathcal{N}_0$ f_i bestimmen.
Dann ist aber $g(n) ::= f_n(n) + 1$ nicht primitiv rekursiv , da sonst gelten müßte:

$f_j(j) = g(j) = f_j(j) + 1$ **Widerspruch!!**

3.3 Die Ackermann-Funktion

Im letzten Abschnitt haben wir gezeigt, daß der folgende Satz gilt:

Satz:
Es gibt Turing-berechenbare Funktionen, die nicht primitiv rekursiv sind.

Bereits 1928 hat **Ackermann** eine Funktion angegeben, die nicht primitiv rekursiv ist.
Zu ihrer Definition verwendete er ein Rekursionsschema, das nicht durch primitive Rekursion formuliert werden kann.
Die Idee ist, daß diese Funktion schneller als alle primitiv rekursiven Funktionen wächst.
Wenn wir die Addition, Multiplikation und Exponentiation betrachten, stellen wir fest, daß alle diese Funktionen nach dem gleichen Schema gebaut sind. Bezeichnen wir die Funktionen mit f_1, f_2, f_3 erhalten wir

$f_n(x, 0) = g_n(x)$
$f_n(x, y') = f_{n-1}(f_n(x, y), x)$

Ackermann machte nun aus dem Index n ein Argument der Funktion

$f(n, x, y) ::= f_n(x, y)$

Damit ergibt sich:

$f(n', x, y') = f(n, f(n', x, y), x)$

Dieses Schema läßt sich nicht primitiv rekursiv formulieren.
Außerdem sieht man, daß das x unverändert "durchgereicht" und nur als Argument von g_n verwendet wird. Also lassen wir x weg und erhalten:

$f(n', y') = f(n, f(n', y))$

Wir benennen n in x um:

$f(x', y') = f(x, f(x', y))$ z.B. $f(1, 1) = f(0, f(1, 0))$.

Für den Rekursionsanfang setzen wir daher:

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

In der Praxis findet diese Funktion Verwendung, um die Effizienz von Prozeduraufrufmechanismen zu testen.
Den formalen Beweis, daß die **Ackermann-Funktion** nicht primitiv rekursiv ist, findet man in [3].
Die primitiv rekursiven Funktionen verwenden im Prinzip nur die Nachfolgerfunktion; dennoch lassen sich mit ihnen nahezu alle interessanten Funktionen beschreiben.
Was man tun muß, um **alle** totalen Turing-berechenbaren Funktionen beschreiben zu können, werden wir im nächsten Abschnitt untersuchen.

3.3.1 Primitiv rekursive Prädikate

Logische Prädikate sind Aussagen, die entweder wahr oder falsch sind.
Äquivalent sind dazu Funktionen, die nur zwei Werte annehmen können, z.B. 0 oder 1; oder etwas allgemeiner den Wert 0 und Werte $\neq 0$.
Wir verknüpfen nun Prädikate und primitiv rekursive Funktionen zu **primitiv rekursiven Prädikaten**

Definition: (primitiv rekursives Prädikat [6])
Wenn eine n-stellige Funktion f primitiv rekursiv ist, so heißt das Prädikat P **primitiv rekursives Prädikat**, wenn gilt:

$$P(x_1, ..., x_n) \Longleftrightarrow f(x_1, ..., x_n) = 0$$

Die Funktion f liefert also genau dann den Wert 0, wenn P gilt.
f bezeichnet man auch als **charakteristische Funktion** von P.

In [3] findet man eine große Kollektion primitiv rekursiver Prädikate; wir sehen uns nur einige davon an:

Gleichheitsbeziehung: $x = y \Longleftrightarrow \varepsilon(x, y) = 0$

Dabei ist $\varepsilon(x, y) = sg(|x - y|)$ mit

$$sg(x) ::= \begin{cases} 0 & : \quad x = 0, sg(0) = 0 \\ 1 & : \quad x > 0, sg(x') = 1 \end{cases}$$

Ähnlich kann man zeigen, daß $\leq, <, \geq, >, /$(Teilbarkeit) primitiv rekursive Prädikate sind.

Jedes primitiv rekursive Prädikat ist entscheidbar, da man ja nur die zugehörige primitiv rekursive Funktion berechnen muß.

Es ist interessant (und hilfreich), daß man aus mehreren primitiv rekursiven Prädikaten weitere Prädikate zusammensetzen kann, die dann auch primitiv rekursiv sind. Wir zeigen, daß man für das zusammengesetze Prädikat eine primitiv rekursive Funktion finden kann, die die Bedingung aus der Definition erfüllt.

Negation :
Sei Q ein primitiv rekursives Prädikat, g die charakteristische Funktion. Dann gilt:
$P(x) \Longleftrightarrow \neg Q(x) \Longleftrightarrow g(x) \neq 0 \Longleftrightarrow \bar{sg}(g(x)) = 0$

Dabei ist

$$\overline{sg}(x) ::= \begin{cases} 1 & : \quad x = 0, \overline{sg}(0) = 1 \\ 0 & : \quad x > 0, \overline{sg}(x') = 0 \end{cases}$$

Konjunktion :
Seien Q, R primitiv rekursive Prädikate, q, r die charakteristischen Funktionen. Dann gilt:
$P(x) \Longleftrightarrow Q(x) \wedge R(x) \Longleftrightarrow q(x) = 0 \wedge r(x) = 0 \Longleftrightarrow q(x) + r(x) = 0$

Alternative :
Seien Q, R primitiv rekursive Prädikate, q, r die charakteristischen Funktionen. Dann gilt:
$P(x) \Longleftrightarrow Q(x) \vee R(x) \Longleftrightarrow q(x) = 0 \vee r(x) = 0 \Longleftrightarrow q(x) * r(x) = 0$

Der Konjunktion entspricht also die **Addition**, der Alternative das **Produkt** der charakteristischen Funktionen.

3.4 μ-rekursive Funktionen

Wenn wir z.B. das Prädikat *x teilt y* betrachten, können wir x festhalten und fragen: *Welches ist das kleinste y, daß von x geteilt wird?*
So erhalten wir eine Funktion, die jedem x das kleinste y zuordnet, das von x geteilt wird. (Nicht besonders interessant, aber wir werden sehen...)
Dieses Minimum-Problem kann man auch in **C** formulieren:

```
y = 0;
while(!p(x,y)) y++;
```

Seltsamerweise kann man dies mit Hilfe primitiv rekursiver Funktionen nicht formulieren.
Die Schleife liefert das erste y_0 ab, für das das Prädikat $p(x, y)$ wahr wird, also die charakteristische Funktion $f(x, y)$ den Wert 0 liefert.
Man beachte, daß $f(x, y)$ für alle Werte $y < y_0$ definiert sein muß.

Definition: (unbeschränkter μ-Operator)
Sei P ein $(n+1)$-stelliges Prädikat und existiere zu $(x_1, ..., x_n)$ ein **eindeutiges, kleinstes** y mit $P(x_1, ..., x_n, y) =$ **true**, dann bezeichnen wir dieses y mit

$$\mu y P(x_1, ..., x_n, y)$$

μ ist der **unbeschränkte** μ-Operator.
Falls ein solches y **nicht** existiert, definieren wir

$$\mu y P(x_1, ..., x_n, y) = 0$$

Zu diesem Prädikat definieren wir f wie folgt:

$$f(x_1, ..., x_n) = \mu y P(x_1, ..., x_n, y)$$

Wenn P entscheidbar ist und ein solches y existiert, ist f Turing-berechenbar.

Dazu testet man einfach $P(x_1, ..., x_n, 0), P(x_1, ..., x_n, 1), P(x_1, ..., x_n, 2), ...$

Dieses Vorgehen bezeichnet man auch als **Anwendung des μ-Operators im Normalfall**.

Falls y nicht existiert, kann es sein, daß die Berechnung nicht abbricht, sonst wird das letzte Argument ausgegeben, sobald die Berechnung den Wert **true** liefert.

Wir bemerken noch, daß die **unbeschränkte Minimalisierung** weder durch primitive Rekursion noch durch Einsetzung formuliert werden kann. Diese beiden Konzepte sind also zu schwach, um die Schleifen höherer Programmiersprachen zu formulieren.

3.4.1 Der eingeschränkte μ-Operator

Definition: (eingeschränkter μ-Operator [6])
Der Ausdruck

$$\mu^y_{z=0} P(\vec{x}, z)$$

definiert das kleinste z zwischen 0 und y einschließlich, für das $P(\vec{x}, z)$ gilt, falls es ein solches z gibt, sonst 0.
Wir sprechen vom **eingeschränkten μ-Operator**.

Aus dem eingeschränkten μ-Operator gewinnt man eine Funktion

$$f(\vec{x}, y) ::= \mu^y_{z=0} P(\vec{x}, z)$$

Im Gegensatz zum allgemeinen μ-Operator hat die Funktion f $(n + 1)$ Stellen.

$\vec{x}$ steht für $(x_1, ..., x_n)$.

Satz:
Ist P ein primitiv rekursives Prädikat, so ist die Funktion f ebenfalls primitiv rekursiv.

Mit Hilfe des eingeschränkten μ-Operators kann man nun zeigen, daß diverse interessante Funktionen primitiv rekursiv sind.

Beispiel:
Die Funktion $p(n)$, die die n-te Primzahl berechnet, ist primitiv rekursiv :

$p(0) = 2$
$p(n') = \mu_{z=0}^{p(n)!+1} Pr(z) \wedge p(n) < z$

Die Eigenschaft $Pr(x)$, daß x Primzahl ist, ist primitiv rekursiv, da man sie aus primitiv rekursiven Prädikaten zusammensetzen kann:

$$Pr(x) \Longleftrightarrow x \neq 0 \wedge x \neq 1 \wedge \forall_{z=0}^{x}(z|x \Rightarrow z = 1 \vee z = x)$$

Im Zusammenhang mit der Gödelisierung werden wir noch die Funktion $exp(n, x)$ benötigen, die angibt, mit welchem Exponenten die n-te Primzahl $p(n)$ in x vorkommt.
Auch diese Funktion können wir mit dem eingeschränkten μ-Operator formulieren:

$$exp(n, x) = \mu_{z=0}^{x} \neg(p(n)^{z+1}|x)$$

Die Funktion berechnet also das kleinste z, so daß x von $p(n)^{z+1}$ nicht mehr geteilt wird.

3.4.2 μ-rekursive Funktionen

Wir werden nun die Anwendung des unbeschränkten μ-Operators im Normalfall zu den Konstruktionsprinzipien der primitiv rekursiven Funktionen hinzufügen.

Definition: (μ-rekursive Funktionen [6])
Eine Funktion heißt μ-rekursiv, wenn sie aus den drei Grundfunktionen der primitiv rekursiven Funktionen, durch Einsetzung, primitive Rekursion oder durch Anwendung des μ-Operators im Normalfall aufgebaut ist.

μ-rekursive Prädikate definieren wir analog zu primitiv rekursiven Prädikaten.

μ-rekursive Funktionen sind Turing-berechenbar .

μ-rekursive Prädikate sind entscheidbar.

μ-Rekursivität ist "mehr" als primitive Rekursivität, wie der folgende Satz zeigt.

Satz:
Die Ackermann-Funktion ist μ-rekursiv.

Beweisskizze:
Der Beweis findet sich natürlich in [3]. Wir skizzieren die Idee. Zunächst betrachten wir die Berechnungsschritte für $f(2,1)$:

0.	f(2,1)	2,1
1.	f(1,f(2,0))	1,2,0
2.	f(1,f(1,1))	1,1,1
3.	f(1,f(0,f(1,0)))	1,0,1,0
4.	f(1,f(0,f(0,1)))	1,0,0,1
5.	f(1,f(0,2))	1,0,2
6.	f(1,3)	1,3
7.	f(0,f(1,2))	0,1,2
8.	f(0,f(0,f(1,1)))	0,0,1,1
9.	f(0,f(0,f(0,f(1,0))))	0,0,0,1,0
10.	f(0,f(0,f(0,f(0,1))))	0,0,0,0,1
11.	f(0,f(0,f(0,2)))	0,0,0,2
12.	f(0,f(0,3))	0,0,3
13.	f(0,4)	0,4
14.	5	5

Die einzelnen Zahlenfolgen erhöhen wir nun alle jeweils um 1, um von 0 verschiedene Werte zu erhalten und verwenden sie als Exponenten der Primzahlen. (Man erinnere sich an die Codierung der Turing-Maschinen)

0.	f(2,1)	2,1	$2^3 3^2 =$	72
1.	f(1,f(2,0))	1,2,0	$2^2 3^3 5^1 =$	540
2.	f(1,f(1,1))	1,1,1		900
3.	f(1,f(0,f(1,0)))	1,0,1,0		2100
4.	f(1,f(0,f(0,1)))	1,0,0,1		2940
5.	f(1,f(0,2))	1,0,2		1500
6.	f(1,3)	1,3		324
7.	f(0,f(1,2))	0,1,2		2250
8.	f(0,f(0,f(1,1)))	0,0,1,1		7350
9.	f(0,f(0,f(0,f(1,0))))	0,0,0,1,0		16170
10.	f(0,f(0,f(0,f(0,1))))	0,0,0,0,1		25410
11.	f(0,f(0,f(0,2)))	0,0,0,2		10290
12.	f(0,f(0,3))	0,0,3		3750
13.	f(0,4)	0,4		486
14.	5	5		64

Aus einer Gödelnummer kann man nun die entsprechende Konstantenfolge berechnen. So erhalten wir eine Funktion $g(x,y,z)$, wobei x,y die Argumente der Ackermann-Funktion und z die Nummer des Berechnungsschritts sind.
Die Schritte nach dem Abschluß der Berechnung setzen wir konstant auf den letzten Wert — im Beispiel also 64.

Man kann nun zeigen, daß $g(x, y, z)$ primitiv rekursiv ist.

Dazu machen wir eine Fallunterscheidung:

$g(x, y, 0)$ dies ist die Gödelnummer des 0-ten Berechnungsschritts, also der Folge x, y und damit $2^{x+1}3^{y+1}$.

Damit haben wir den Anfang und müssen nun den Wert von $g(x, y, z')$ aus $g(x, y, z)$ berechnen. Dazu berechnen wir aus $g(x, y, z)$ die Folge $x_1, ..., x_n$ und unterscheiden vier Fälle:

1. $n = 1$: Dies kann nur der Wert des letzten Schritts sein. Also setzen wir $g(x, y, z') = g(x, y, z)$, da ja nach diesem Schritt $g(x, y, z)$ konstant sein soll.
2. $n > 1$ und $x_{n-1} = 0$: dies entspricht dem Fall $f(0, y)$ als letztem Glied in der Funktionsschachtelung $f(...f(0, y)...)$. Den nächsten Wert erhalten wir durch "Vergödeln" von $x_1, ..., x_{n-2}, x_n{}'$.
3. $n > 1$ und $x_n = 0$: dies entspricht dem Fall $f(x', 0)$ als letztem Glied in der Funktionsschachtelung $f(...f(x', 0)...)$. Den nächsten Wert erhalten wir durch "Vergödeln" von $x_1, ..., x_{n-2}, V(x_{n-1}), 1$. V ist die primitiv rekursive Vorgängerfunktion.
4. $n > 1$ und $x_{n-1} \neq 0$ und $x_n \neq 0$: dies ist der Fall $f(x', y')$ als letztem Glied in der Funktionsschachtelung $f(...f(x', y')...)$. Den nächsten Wert erhalten wir durch "Vergödeln" von $x_1, ..., x_{n-2}, V(x_{n-1}), x_{n-1}, V(x_n)$.

Der letzte Schritt n in der Berechnung, ist nun der Schritt, ab dem g konstant bleibt. Und das kann man mit dem μ-Operator formulieren:

$$n = \mu z[g(x, y, z) = g(x, y, z')]$$

Der Wert von g im Schritt z ist ist derselbe wie im Schritt z', und n ist das kleinste z mit dieser Eigenschaft. Wir können nun n als Funktion von x und y auffassen:

$$f_S(x, y) ::= \mu z[g(x, y, z) = g(x, y, z')]$$

Da g primitiv rekursiv ist, ist f_S μ-rekursiv.

$g(x, y, f_S(x, y))$ ist das Ergebnis von g nach dem letzten Berechnungsschritt der Ackermann-Funktion f, also die Gödelnummer des Ergebnisses (von f). Um dieses Ergebnis zu erhalten, müssen wir mit der Funktion exp den Exponenten bestimmen, als der das Ergebnis auftritt, und davon mit V den Vorgänger berechnen.

$$f(x, y) = V(exp(0, g(x, y, f_S(x, y))))$$
$$= V(exp(0, g(x, y, \mu z[g(x, y, z) = g(x, y, z')])))$$

Zum Abschluß noch einige grundlegende Sätze, deren Beweise man in [3] findet.

Satz:
Jede μ-rekursive Funktion ist Turing-berechenbar.

Um dies zu zeigen, stellt man die Realisierung des μ-Operators im Normalfall durch eine Turing-Maschine dar, läßt $\mu z P(\vec{x}, z)$ bei 0 beginnen, wendet die Turing-Maschine , die P entscheidet, darauf an und erhöht z solange, bis P wahr wird.

Satz:
Jede total Turing-berechenbare Funktion ist μ-rekursiv.

Die Beweisidee ist ähnlich zum Beweis der μ-Rekursivität der Ackermann-Funktion.

Diese Äquivalenzen zeigen einmal mehr, daß es sich bei den Turing-berechenbaren Funktionen offenbar genau um die Funktionen handelt, die man irgendwie mechanisch berechnen kann.
Und das hatten wir in der **Church-Turing-These** ja auch schon festgestellt.

Kapitel 4

Komplexitätstheorie

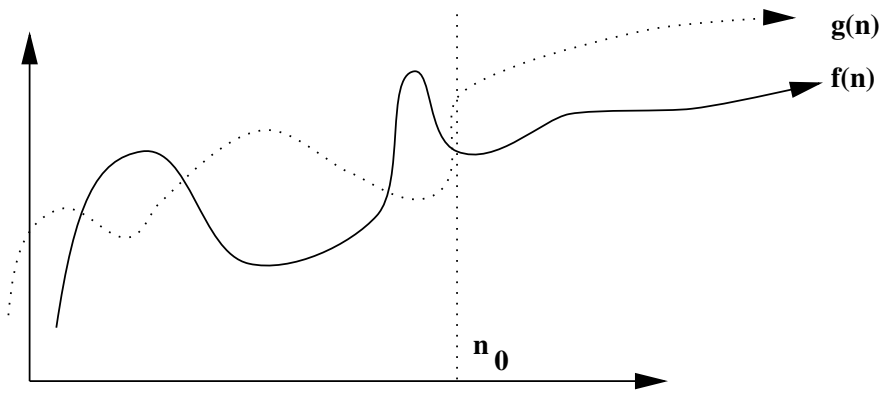
Bisher haben wir uns mit der Frage beschäftigt, ob es für die Lösung bestimmter Probleme **überhaupt** einen Algorithmus gibt. Dabei haben wir uns nicht um die **Effizienz** gekümmert.
Im folgenden Kapitel wollen wir untersuchen, ob Algorithmen mit "vernünftigem" Aufwand ablaufen, was ihr Zeit- bzw. Speicherplatzverhalten angeht. Wobei noch zu klären ist, was in diesem Zusammenhang "vernünftig" bedeutet.

4.1 Die Groß-O-Notation

In den *Algorithmen und Datenstrukturen* haben Sie sich bereits mit der Effizienz von Algorithmen beschäftigt. Zur Erinnerung

Definition: (Groß-O-Notation)
$O(g(n)) = \{f | \exists c > 0, n_0 > 0 : \forall n > n_0 : f(n) \leq cg(n)\}$

Für große n wird das Verhalten von $f(n)$ also durch $g(n)$ bestimmt.



Diese Notation wurde verwendet, um **obere Schranken** für die Komplexität von Algorithmen anzugeben. Wir erinnern uns an einige **Eigenschaften**:

$f \in O(g(n)), g \in O(h(n)) \Rightarrow f \in O(h(n))$

$O(f + g) = O(f) + O(g) = O(max(f, g))$

Beispiele:

Sequentielle Suche	$O(n)$
Binäre Suche	$O(log\ n)$
Quicksort	$O(n\ log\ n)$
Multiplikation von 2 n-stelligen Zahlen	$O(n^2)$
Matrixmultiplikation	$O(n^3)$
Travelling Salesman	$O(2^n)$

Angenommen ein Algorithmus benötigt 10^{-6} Sekunden für eine Operation. Dann ergibt sich folgende Übersicht:

n	$O(log_2 n)$	$0(n)$	$O(n^2)$	$O(2^n)$
10	0.000003 sec	0.00001 sec	0.0001 sec	0.001 sec
100	0.000007 sec	0.0001 sec	0.01 sec	10^{16} y
1000	0.00001 sec	0.001 sec	1.0 sec	astronomisch
10000	0.000013 sec	0.01 sec	1.7 min	”
100000	0.000017 sec	0.1 sec	2.8 h	”

Dabei setzen wir $2^{100} \approx 10^{30}, 1y \approx 3 * 10^7$ sec.

Man sieht, daß die Rechenzeit bei Algorithmen aus $O(2^n)$ nahezu explodiert, während sie sich bei den anderen Schranken noch in Grenzen hält.

Der **Zeitbedarf eines Algorithmus** f $min\{g|f \in O(g(n))\}$ kennzeichnet das Zeitverhalten von $f(n)$ für n Eingaben.

Die **Komplexität** eines Algorithmus mit einem Zeitbedarf $f(n)$ heißt

- **polynomial** , wenn $f \in O(n^c)$
- **exponentiell** , wenn $f \in O(c^n)$

Algorithmen mit polynomialem Verhalten sind oft durchführbar, während exponentielle Algorithmen nur für **sehr kleine** Eingabemengen durchführbar sind. Leider führen aber gerade wichtige Planungsprobleme zu exponentiellen Algoithmen.

Im nächsten Abschnitt sehen wir uns ein wichtiges Problem an, für das kein Algorithmus bekannt ist, der in polynomialer Zeit arbeitet.

4.2 Das Erfüllbarkeitsproblem

Dieses Problem läßt sich sehr einfach beschreiben:

Gegeben ist ein Boolescher Ausdruck mit n Variablen. Gesucht ist eine Belegung dieser Variablen, so daß der Ausdruck den Wert **true** erhält.

Beispiel:

$A = (x_1 + \overline{x_3} + x_4)(\overline{x_1} + \overline{x_2} + \overline{x_4})(\overline{x_2} + x_3)(\overline{x_1} + x_2 + x_4)$

(A bezeichnet man als Summenproduktausdruck.)

$x_1 = 0, x_2 = x_3 = x_4 = 1$ erfüllt die Bedingung.

Eine naheliegende Lösungsmöglichkeit ist, einfach alle 2^n Möglichkeiten durchzuprobieren. Ist das Problem nicht erfüllbar, so merkt man das allerdings erst nach 2^n Schritten.

In den 60er Jahren haben **Davis** und **Putnam** einen Algorithmus gefunden, der in vielen Fällen schneller zum Ziel führt, **falls** der Ausdruck erfüllbar ist:

procedure *Aufteilen(A)*

1. wenn A eine **leere Klausel** enthält return
2. wenn A **keine Klausel** mehr enthält exit mit aktueller Zuordnung
3. Wähle nächste, nicht belegte Variable x_i aus A.
4. *Aufteilen*$(A(x_i = 0))$
5. *Aufteilen*$(A(x_i = 1))$

Im Ausdruck $A(x_i = 0)$ werden die Klauseln entfernt, die $\overline{x_i}$ enthalten. Aus allen Klauseln wird x_i entfernt.
Im Ausdruck $A(x_i = 1)$ werden ganz analog die Klauseln entfernt, die x_i enthalten. Aus allen Klauseln wird $\overline{x_i}$ entfernt.
Entstehen dabei **leere** Klauseln, so führt die aktuelle Teilbelegung nicht zum Ziel.

Beispiel:

$A = (x_1 + \overline{x_3} + x_4)(\overline{x_1} + \overline{x_2} + \overline{x_4})(\overline{x_2} + x_3)(\overline{x_1} + x_2 + x_4)$

$A(x_1 = 0) = (\overline{x_3} + x_4)(\overline{x_2} + x_3)$

$A(x_1 = 1) = (\overline{x_2} + \overline{x_4})(\overline{x_2} + x_3)(x_2 + x_4)$

$A(x_1 = 0, x_2 = 0) = (\overline{x_3} + x_4)$

$A(x_1 = 1, x_2 = 0) = (x_4)$

$A(x_1 = 0, x_2 = 1) = (\overline{x_3} + x_4)(x_3)$

$A(x_1 = 1, x_2 = 1) = (\overline{x_4})(x_3)$

$A(x_1 = 0, x_2 = 0, x_3 = 0) = \emptyset$ *

$A(x_1 = 1, x_2 = 0, x_3 = 0) = (x_4)$

$A(x_1 = 0, x_2 = 1, x_3 = 0) = (\emptyset)$ **

$A(x_1 = 1, x_2 = 1, x_3 = 0) = (\overline{x_4})(\emptyset)$

$A(x_1 = 0, x_2 = 0, x_3 = 1) = (x_4)$

$A(x_1 = 1, x_2 = 0, x_3 = 1) = (x_4)$

$A(x_1 = 0, x_2 = 1, x_3 = 1) = (x_4)$

$A(x_1 = 1, x_2 = 1, x_3 = 1) = (\overline{x_4})$

...

* liefert also bereits ein Ergebnis, während ** eine unzulässige Belegung darstellt.

In vielen Fällen findet der Algorithmus eine Lösung in wesentlich kürzerer Zeit als die erschöpfende Suche. Aber es kann nicht **garantiert** werden, daß das Problem in weniger als exponentieller Zeit gelöst wird.

Eine einfacher aussehende Version des Problems scheint aber genauso schwierig zu sein: gesucht ist ein polynomialer Algorithmus, der für jeden Summenproduktausdruck, der erfüllbar ist, eine 1, sonst eine 0 ausgibt.

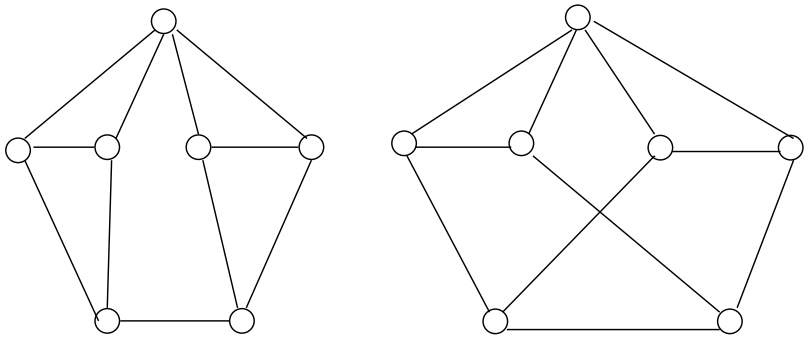
Hier möchte man also nur wissen, ob eine erfüllende Zuordnung **existiert**.

Es wird sich zeigen, daß dieses **Erfüllbarkeits-Entscheidungsproblem** von zentraler Bedeutung ist, da sich viele Probleme darauf zurückführen lassen.

4.3 Färbung von Graphen

Betrachten wir das Problem aus dem letzten Abschnitt in etwas anderer Verkleidung:

Gegeben ist ein Graph mit n Knoten. Wir suchen eine **Einfärbung** der Knoten mit maximal drei Farben, bei der benachbarte Knoten verschiedene Farben haben.



Bei einem der obigen Graphen gibt es eine Färbung, beim anderen nicht.

Die Frage ist, existiert ein Algorithmus, der diese Färbung in polynomialer Zeit berechnet?

Es zeigt sich, daß dieses Problem mit dem Erfüllbarkeitsproblem eng verwandt ist. Man kann nämlich einen Algorithmus angeben, der dieses Problem in n^2 Schritten in einen **Summenproduktausdruck** umwandelt, der genau dann erfüllt ist, wenn der Graph mit drei Farben zulässig färbbar ist.

Die Existenz dieses Algorithmus bedeutet, daß das Erfüllungs-Entscheidungsproblem mindestens genauso schwierig ist, wie das Dreifarben-Entscheidungsproblem.
Der Algorithmus sieht wie folgt aus:

procedure *Transform*

1. für jeden Knoten v_i
   Ausgabe: $(r_i + g_i + b_i)$
2. für jede Kante (v_i, v_j)
   Ausgabe:$(\overline{r_i} + \overline{r_j})(\overline{b_i} + \overline{b_j})(\overline{g_i} + \overline{g_j})$

Das sind maximal $n + n(n-1)$ Faktoren.

r_i hat den Wert 1, wenn Knoten v_i **rot** ist, 0 sonst.
b_i hat den Wert 1, wenn Knoten v_i **blau** ist, 0 sonst.
g_i hat den Wert 1, wenn Knoten v_i **grün** ist, 0 sonst.

In einer zulässigen Färbung erhält jeder Knoten eine Farbe, also sind die Faktoren $(r_i + g_i + b_i)$ wahr. Zwei benachbarte Knoten sind verschieden gefärbt, daher sind die Faktoren aus (2) wahr.
Findet man eine erfüllende Belegung für die Variablen im Summenproduktausdruck, so ist auch das Dreifarbenproblem gelöst.

Nach diesen eher praktischen Betrachtungen werden wir erst einmal wieder etwas Theorie treiben und uns im nächsten Abschnitt einer Klasse von Problemen zuwenden, die man als **prinzipiell schwierig** ansehen kann.

4.4 Komplexitätsklassen

Wir werden unsere Entscheidugsprobleme in zwei Klassen einteilen:
einmal sind da die Probleme, die man in **polynomialer Zeit** mit **determinisischen** Algorithmen lösen kann, also die, für deren Lösung ein (mehr oder weniger) effizientes Verfahren existiert.

Die andere Klasse enthält die schwierigen Probleme, für die nur **nichtdeterministische** Algorithmen existieren, die eine Lösung in polynomialer Zeit liefern. Formaler ausgedrückt:

Definition:
$\mathcal{P} = \{L | \exists$deterministische TM T : $L = L(T),\ O(x \in L)$ polynomial$\}$
$\mathcal{NP} = \{L | \exists$**nicht**deterministische TM T : $L = L(T),\ O(x \in L)$ polynomial$\}$

Äquivalent dazu kann man sagen, $\mathcal{P}$ enthält die Probleme, die sich in polynomialer Zeit mit deterministischen Algorithmen **lösen** lassen, $\mathcal{NP}$ die Probleme, deren **Lösung** man deterministisch in polynomialer Zeit **verifizieren** kann.

Definition:
$A, B \subset E^*$; A heißt **polynomial auf B reduzierbar**, $A \leq_P B$, wenn es eine totale, mit polynomieller Komplexität berechenbare Funktion $f : E^* \to E^*$ gibt, mit $\forall x \in E^* : x \in A \Leftrightarrow f(x) \in B$

4.5 NP-Vollständigkeit

Wir nennen eine Sprache $A \in \mathcal{NP}$ **NP-vollständig** ,wenn für alle Sprachen $L \in \mathcal{NP}$ gilt, $L \leq_P A$.

Das bedeutet, daß A genauso schwierig ist, wie jedes andere Problem aus $\mathcal{NP}$.

Praktisch gesehen heißt das, ein NP-vollständiges Problem ist ein Problem, das auf dem Computer nur lösbar ist, wenn man gewillt ist, **extrem lange** auf die Lösung zu warten.

1970 hat **Steven Cook** entdeckt, daß das **Erfüllbarkeitsproblem** NP-vollständig ist. Er fand eine Transformation, die jedes Problem aus $\mathcal{NP}$ auf dieses Problem abbildet.

Es ist bis heute nicht bekannt, ob es wirklich Probleme gibt, für die es keine Algorithmen aus $\mathcal{P}$ gibt. Man vermutet, daß $\mathcal{P} \subset \mathcal{NP}$ gilt, daß die Inklusion also echt ist. Aber bewiesen ist das nicht.

Die NP-vollständigen Probleme spielen hier eine wichtige Rolle:

könnte man von einem NP-vollständigen Problem **beweisen**, daß es aus $\mathcal{P}$ ($\mathcal{NP}$) ist, so hätte man den Beweis dafür, daß $\mathcal{P} = \mathcal{NP}$ ($\mathcal{P} \neq \mathcal{NP}$) gilt.

4.6 Der Satz von Cook

Wir wollen den Beweis, daß das Erfüllungsproblem (engl. **SAT** von satisfiability) NP-vollständig ist, hier nur kurz skizzieren. Um diese Behauptung zu beweisen, muß man alle Probleme aus $\mathcal{NP}$ in polynomialer Zeit auf **SAT** abbilden können.

Wenn ein Problem aus $\mathcal{NP}$ ist, so existiert eine nichtdeterministische Turing-Maschine T, die das Problem in polynomialer Zeit löst. Das kann man sich auch so vorstellen, daß die Maschine eine Anzahl Lösungsmöglichkeiten **rät**, und diese dann **verifiziert**.

Die Grundidee ist nun, die Arbeitsweise einer solchen Turing-Maschine durch logische Ausdrücke zu beschreiben, die genau dann erfüllbar sind, wenn die Turing-Maschine eine Lösung findet.

T macht zur Lösung eines Problems $p(n)$ Schritte, wobei $p(n)$ ein Polynom in n darstellt.

Zu jedem Zeitpunkt kann T sich in **genau einem Zustand** befinden. Das Prädikat $S(t, z)$ soll genau dann wahr sein, wenn T sich zum Zeitpunkt t im Zustand z befindet. Also muß für T gelten:

$$\forall t \in \{0, 1, ..., p(n)\}, \forall z, z' \in Z : z \neq z' \Rightarrow (S(t, z) \Rightarrow \neg S(t, z'))$$

Dies liefert uns eine Konjunktion aus $(p(n) + 1) * |Z| * (|Z| - 1)$ Klauseln, die man in disjunktiver Normalform hinschreiben kann.

Es müssen weitere Bedingungen gelten:
Jedes Feld des Bandes kann **genau ein** Zeichen aufnehmen:
Das Prädikat $T(t, c, s)$ ist genau dann wahr, wenn zur Zeit t das Feld c das Symbol s enthält. Also muß gelten:

$$\forall t \in \{0, 1, ..., p(n)\}, \forall s \neq s' \in B, -p(n) \leq c \leq p(n)(T(t, c, s) \Rightarrow \neg T(t, c, s'))$$

Das liefert uns weitere $(p(n) + 1) * (2p(n) + 1) * |B| * (|B| - 1)$ Klauseln.

Zu einem Zeitpunkt kann sich der Kopf der Turing-Maschine nur über einem Feld befinden. $H(t, c)$ ist wahr, wenn der Kopf zum Zeitpunkt t auf Feld c steht, also:

$$\forall t \in \{0, 1, ..., p(n)\}, -p(n) \leq c \neq c' \leq p(n) \Rightarrow (H(t, c) \Rightarrow \neg H(t, c'))$$

Das liefert uns $(p(n) + 1) * (2p(n) + 1)$ Klauseln.

Es kann auch immer nur das Feld verändert werden, über dem sich der Kopf befindet:

$$\forall t \in \{0, 1, ..., p(n)\} \forall s \in B, -p(n) \leq c \leq p(n)$$

$$((T(t, c, s) \wedge \neg H(t, c)) \Rightarrow T(t + 1, c, s))$$

Das liefert uns weitere $(p(n) + 1) * (2p(n) + 1) * |B|$ Klauseln.

Zum Zeitpunkt 0 befindet sich T auf Feld 0: $H(0, 0)$
Zum Zeitpunkt 0 befindet sich T im Zustand 0: $S(0, 0)$
In den Feldern $s_1, ..., s_n$ befindet sich das Eingabewort:

$$T(0, i, s_i), 1 \leq i \leq n$$

Wenn wir annehmen, daß T m Zustände hat, können wir m als Endzustand annehmen:

$$S(p(n), m)$$

Zur zeit $p(n)$ soll T mit "ja" antworten, d.h. auf Feld 0 steht eine 1:

$$T(p(n), 0, 1)$$

Nun müssen noch die Zustandübergänge codiert werden. Das läßt sich aus der Turing-Tafel ablesen:

$$(z, s \rightarrow z', s', d), d \in \{L, R\}$$

Für $t = 0, 1, ..., p(n), c = -p(n), ..., p(n), z \in Z, s \in B$ codieren wir

$$(S(t, z) \wedge T(t, c, s) \wedge H(t, c)) \Rightarrow S(t + 1, z')$$

$$(S(t, z) \wedge T(t, c, s) \wedge H(t, c)) \Rightarrow T(t + 1, c, s')$$

$$(S(t, z) \wedge T(t, c, s) \wedge H(t, c)) \Rightarrow H(t + 1, c')$$

Die Menge dieser Klauseln, die sich in polynomialer Zeit berechnen lassen, liefert uns einen logischen Ausdruck, der genau dann erfüllbar ist, wenn die Turing-Maschine auf die Frage: *liegt eine Lösung des Problems vor?* mit "ja" antwortet.

4.7 NP-vollständige Probleme

Um von einem Problem sagen zu können, daß es prinipiell schwierig zu lösen sein wird, muß man nachweisen, daß es sich auf ein bereits bekanntes, NP-vollständiges Problem reduzieren läßt.

Ist dies der Fall, so ist die Entwicklung eines effizenten Algorithmus eher unwahrscheinlich.

Es gibt eine ganze Anzahl praktisch relevanter Probleme, die NP-vollständig sind:

- **Travelling Salesman**
  auch als *Problem des Handlungsreisenden* bekannt: man finde eine kürzeste Rundreise durch eine gegebene Zahl von Städten, die genau einmal besucht werden.
- **Hamilton Zyklus**
  existiert zu einer gegebenen Karte eine Rundreise, bei der jeder Ort genau einmal besucht wird.
- **Stundenplanung**
  das für einen **Prodekan** wichtigste Problem
- **Rucksackproblem**
  gegeben sind eine Anzahl von Waren verschiedener Größe und von unterschiedlichem Wert. Man belade einen Wagen (Rucksack, ...) so, daß man Waren von maximalem Gesamtwert transportiert.

Index

μ-Operator, 89
  eingeschränkter, 91
  unbeschränkter, 89
μ-rekursive Funktion, 93
μ-rekursives Prädikat, 93
$\mathcal{L}_{TM}$, 33
$\mathcal{L}_{ndetKA}$, 33
Überführungsfunktion, 19
Übergangsrelation, 24
überabzählbar, 8

absolut entscheidbar, 10
Abu Jaf'ar Mohammed ibn Mûsâ al-Khowârizm, 2
abzählbar, 13
Abzählbarkeit, 8
Ackermann, 83
Ackermann-Funktion, 83, 84
Addition, 76, 79
akzeptieren, 29
Akzeptor, 27
Alan Turing, 17
Algorithmus, 1, 2, 5, 17
Alphabet, 5
Alternative, 87
Anfangszustand, 19
aufzählbar, 13
aufzählbare Menge, 12
Aufzählbarkeit, 12, 57
Aufzählverfahren, 6, 12
Automat
  endlicher, 18, 27

Band, 18
Bandalphabet, 19
Bandinschrift, 23
Basisfunktionen, 77
berechenbar, 7, 48
berechenbare Funktion, 7, 17
Berechenbarkeit, 1
bijektiv, 8
Binäre Suche, 101
busy beaver Problem, 50

charakteristische Funktion, 11, 85
Church-Turing-These, 56, 98
Cook
  Satz von, 111
Cook, Steven, 110

Davis, 104
DEA, 30

Einfärbung, 107
Eingabealphabet, 19
Einsetzung, 77
endlicher Automat, 18, 27
endscheidbar, 13
Endzustände, 19
Enigma, 17
entscheidbar, 10
Entscheidbarkeit, 1, 57
Entscheidungsverfahren, 6
Erfüllbarkeits-Entscheidungsproblem, 106
Erfüllbarkeitsproblem, 103
Erfüllungs-Entscheidungsproblem, 108
Euklidischer Algorithmus, 2
Exponentiation, 76
exponentiell, 102

Felder, 18
Finalkonfiguration, 27, 81
fleißiger Biber, 50
Funktion
  μ-rekursive, 93
  berechenbare, 7, 17
  charakteristische, 11

Gödel, Kurt, 72
Gödelisierung, 72
Gödelnummer, 73, 75, 95
ggt, 2
Green, M.W., 55
Groß-O-Notation, 100
große Linksmaschine, 43
große Rechtsmaschine, 43

Halteproblem, 58, 65
Hamilton Zyklus, 114

Initialkonfiguration, 27

Kellerautomat, 17, 27
Kitab al-jabr w'al-muqabala, 2
Komplexität, 102
Komplexitätsklassen, 109
Konfiguration, 24
Konjunktion, 87
konstante Funktion, 77
kontexsensitiv, 68
Kontrolleinheit, 18
Kopiermaschine, 45

Lösung, 5
lexikographische Ordnung, 8
lineare Bandbeschränkung, 68
Links-Suchmaschine, 44
Linksmaschine, 35, 36

Matrixmultiplikation, 101
Menge
    aufzählbare, 12
Multiplikation, 76, 101

Nachfolgerfunktion, 76, 77, 79
Negation, 87
Nichtentscheidbarkeit, 58
NP-vollständig, 110
NP-Vollständigkeit, 110

O, 100

Péter, 82
Pfeildiagramm, 37
Polynome, 80
polynomial, 102
Prädikat, 85
    mu-rekursives, 93
    primitiv rekursives, 85
primitiv rekursiv, 77
primitiv rekursives Prädikat, 85
primitive Rekursion, 78
Problemausprägungen, 5
Problemklasse, 5
Prodekan, 114
Projektionsfunktion, 77
Putnam, 104

Quicksort, 101

Rado, 55
Rado, Tibor, 50
Rechts-Suchmaschine, 43
Rechtsmaschine, 35, 36
Rekursion, 77
Rucksackproblem, 114

SAT, 111
Schreib/Lesekopf, 18
Sequentielle Suche, 101
simulieren, 58
Speicherplatzverhalten, 99
Sprache, 29
Startmaschine, 37
Stundenplanung, 114
Suchmaschine, 44
Summenproduktausdruck, 107

terminieren, 4
Translationsmaschine, 46
Travelling Salesman, 101, 114
Turing, Alan, 17
Turing-aufzählbar, 57
Turing-berechenbar, 93
Turing-berechenbare Funktionen, 81
Turing-Berechenbarkeit, 48
Turing-entscheidbar, 57
Turing-Maschine, 17, 18
Turing-Tafel, 20
Typ-0, 68
Typ-0-Grammatik, 10, 34
Typ-1, 68
Typ-2, 68

Universelle Turing-Maschine, 58
UTM, 58

verifizieren, 109
Verschiebemaschine, 47
Vorgängerfunktion, 80

Zeichenmaschine, 36
Zeichenmaschinen, 35
Zeitbedarf, 102
Zeitverhalten, 99
Zustandsmenge, 19

Literaturverzeichnis

[1] Th. et al. Cormen. *Introduction to Algorithms.* MIT Press, Cambridge, London, 1983.

[2] A.K. Dewdney. *Der Turing Omnibus.* Springer, Berlin Heidelberg New York, 1995.

[3] Hans Hermes. *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit.* Springer, Berlin, Heidelberg, New York, 1971.

[4] A. Hodges. *Alan Turing: the enigma.* Vintage, London, 1983.

[5] P. Ottmann, T. und Widmayer. *Algorithmen und Datenstrukturen.* BI-Wissenschaftverlag, Mannheim, 1990.

[6] B. Owsnicki-Klewe. *Grundlagen der Berechenbarkeits- und Komplexitätstheorie.* Vorlesungsskript FH-Hamburg, Hamburg, 1996.

[7] Rodger Penrose. *Computerdenken.* Spektrum der Wissenschaft, Heidelberg, 1991.

[8] John E. Hopcroft und Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley (Deutschland) GmbH, Bonn, 1994.

[9] P.Sander und W.Stucky und R.Herschel. *Automaten Sprachen Berechenbarkeit.* Teubner, Stuttgart, 1995.