Formale Grundlagen der Informatik 2
(Theoretische Informatik)

(Skriptum WS 2002/2003)

Franz Winkler

Institut für Symbolisches Rechnen
(RISC-Linz)
Johannes Kepler Universität
Linz

1. Algorithmen

1.1. Was ist ein Algorithmus?

Der Begriff des Algorithmus ist in der Mathematik von alters her gebräuchlich. Intuitiv verstehen wir darunter ein Verfahren, welches von einer Anzahl von Grunddaten (Eingabe des Algorithmus) ausgeht und nach einer endlichen Anzahl wohldefinierter, effektiver Schritte eine Anzahl von Resultaten (Ausgabe des Algorithmus) liefert. Als Urahn aller Algorithmen wird vielfach der *Euklid'sche Algorithmus* zur Berechnung des größten gemeinsamen Teilers ganzer Zahlen angesehen. Man findet ihn bereits im 7. Buch der *"Elemente"* von Euklid ($\sim$ 300 v.Chr.), viele Gelehrte vermuten jedoch, daß es sich dabei eigentlich um Euklid's Fassung eines auf Eudoxus ($\sim$ 375 v.Chr.) zurückgehenden Algorithmus handelt.

Euklid'scher Algorithmus (moderne Fassung):
Zu zwei gegebenen nichtnegativen ganzen Zahlen m und n findet der Algorithmus ihren größten gemeinsamen Teiler (ggt).

(1) $[n = 0?]$ Ist $n = 0$, so endet der Algorithmus mit m als Resultat.

(2) $[m \pmod n]$ Setze $l \leftarrow m \pmod n$, $m \leftarrow n$, $n \leftarrow l$ und fahre mit Schritt (1) fort. ■

Die Korrektheit des Algorithmus sieht man unmittelbar aus der Beziehung

$$\mathrm{ggt}(m, n) = \mathrm{ggt}(n, m - q \cdot n),$$

wobei q eine beliebige ganze Zahl ist.

Als Beispiel dafür berechnen wir etwa den größten gemeinsamen Teiler der beiden Zahlen 40902 und 24140:

$$\begin{aligned}
\mathrm{ggt}(40902, 24140) &= \mathrm{ggt}(24140, 16762) = \mathrm{ggt}(16762, 7378)\\
&= \mathrm{ggt}(7378, 2006) = \mathrm{ggt}(2006, 1360)\\
&= \mathrm{ggt}(1360, 646) = \mathrm{ggt}(646, 68)\\
&= \mathrm{ggt}(68, 34) = \mathrm{ggt}(34, 0)\\
&= 34.
\end{aligned}$$

Im Folgenden wollen wir einige Aspekte des *informalen* Begriffs *"Algorithmus"* und *"durch einen Algorithmus berechenbare Funktion"* zusammenstellen. Oft beschränken wir uns auf Algorithmen, welche als Eingabe ein k–Tupel natürlicher Zahlen nehmen (für ein festes k) und als Ausgabe eine natürliche Zahl liefern. Ein Algorithmus berechnet also eine zahlentheoretische Funktion. Praktisch ist das keine Einschränkung, da alle interessanten und vernünftigen Ein– und Ausgabemengen in den natürlichen Zahlen kodiert werden können.

Es ist wichtig zu unterscheiden zwischen dem Begriff des *Algorithmus*, d.h. der Rechenvorschrift, Prozedur, und dem Begriff der *durch einen Algorithmus berechenbaren Funktion*, d.h. der durch eine Rechenvorschrift gegebenen Abbildung. Ein und dieselbe Funktion kann durch viele verschiedene Algorithmen gegeben sein.

Einige Beispiele für Funktionen, für die ein Algorithmus bekannt ist:

a. $\lambda x, y.x + y$.

b. $\lambda x.(x-$te Primzahl). Ein Algorithmus ist zum Beispiel das *Sieb des Eratosthenes*.

c. $\lambda x, y.\mathrm{ggt}(x, y)$. Dazu kann man den *Euklidschen Algorithmus* benutzen.

d. $\lambda x.$(die natürliche Zahl ≤ 9, die als x–te Ziffer in der Dezimaldarstellung von $\pi = 3.14159\ldots$ vorkommt). Etwa durch Quadratur des Kreises mittels der *Simpson Regel*.

Def. 1.1.1: (informal) Wir führen einige Eigenschaften an, welche man üblicherweise mit dem informalen Begriff des *Algorithmus* verbindet. Davon sind die Eigenschaften (1) — (5) praktisch von allen Mathematikern und Computerwissenschaftern als notwendig anerkannt, über (6) — (10) gibt es gelegentlich verschiedene Auffassungen.

(1) Ein Algorithmus besteht aus einer endlichen Anzahl von Anweisungen oder Instruktionen.

(2) Es gibt einen *Rechner*, Mensch oder Maschine, der diese Instruktionen ausführen kann.

(3) Es besteht die Möglichkeit, Rechenschritte abzuspeichern und aus dem Speicher wieder abzurufen.

(4) Sei P eine Menge von Instruktionen wie in (1) und R ein Rechner wie in (2). Dann besteht die Abarbeitung von P durch R in einem diskreten, schrittweisen

Rechenvorgang (keine analogen Methoden).

(5) Die Abarbeitung von P durch R ist deterministisch, ohne Bezugnahme auf Zufallsmethoden, z.B. Würfeln.

(6) Die Länge der Eingabe ist nach oben hin nicht beschränkt. (Die Theorie der Algorithmen beschäftigt sich mit der Frage der prinzipiellen Berechenbarkeit, ohne Bezugnahme auf praktische Einschränkungen.)

(7) Die Anzahl der Instruktionen ist nach oben hin nicht beschränkt. (Siehe (6).)

(8) Der verfügbare Speicherplatz ist nach oben hin nicht beschränkt. (Siehe (6).)

(9) Der Rechner R kann nur eine endliche Anzahl von Typen von Instruktionen verarbeiten.

(10) Eine a priori Schranke für die Anzahl der benötigten Rechenschritte muß nicht vorhanden sein. ∎

Dieses intuitive Konzept eines Algorithmus bezieht seine Legitimität aus der vielfach geteilten Überzeugung, daß es möglich ist zu erkennen, ob ein Verfahren effektiv ist oder nicht. Das mag noch angehen, wenn es sich darum handelt von einem vorgelegten Verfahren zu entscheiden, ob es ein Algorithmus ist oder nicht. Die Intuition ist aber von geringem Nutzen, wenn es darum geht von einem gewissen Problem nachzuweisen, daß es keinen wie immer gearteten Algorithmus zu dessen Lösung gibt. So gibt es z.B. keinen Algorithmus, der für zwei beliebig vorgegebene Pascal–Programme entscheiden könnte, ob sie das gleiche Input–Output Verhalten haben! Probleme dieser Art treten sehr häufig in der theoretischen Informatik auf, und haben in den letzten Jahrzehnten besondere Aufmerksamkeit erlangt. Eine wissenschaftlich einwandfreie Definition des Begriffs “Algorithmus” ist also in höchstem Maße wünschenswert.

Ein wesentlicher Anstoß zur exakten Fassung des Begriffs “Algorithmus” kam von David Hilberts Programm, das das Ziel verfolgte, einen Algorithmus zu finden, der es gestatten würde, von jeder logischen Formel über den ganzen Zahlen, etwa

$$\neg\Big(\exists n, x, y, z\Big)\Big(\quad x^n + y^n = z^n,\ x \cdot y \cdot z \neq 0,\ n > 2\Big)$$

(Fermats Letzter Satz [1]) zu entscheiden, ob sie gültig ist oder nicht. Daß dieses

[1] Die Gültigkeit von Fermats Letztem Satz wurde in jüngster Zeit von Andrew Wiles bewiesen. Zuvor war diese Frage mehr als 300 Jahre lang ein offenes Problem.

Vorhaben zum Scheitern verurteilt war, wurde 1931 von Kurt Gödel [2] in seinem Unvollständigkeitssatz nachgewiesen:

In jeder Axiomatisierung der Theorie der ganzen Zahlen gibt es Formeln, deren Gültigkeit sich im System weder beweisen noch widerlegen läßt.

Um 1930 arbeiteten viele Mathematiker an diesem Hilbertschen Projekt, unter ihnen A.M. Turing, E.L. Post, A. Church. Verschiedene Modelle für das intuitive Konzept des Algorithmus wurden entwickelt: Turing–Maschinen von A.M. Turing [3], ein ähnliches Modell von E.L. Post, der λ–Kalkül von A. Church. Es stellte sich jedoch heraus, daß alle diese Modelle dieselbe Berechnungspotenz besitzen.

Im folgenden wollen wir verschiedene Annäherungen an den Begriff der Berechenbarkeit etwas näher betrachten.

[2] K. Gödel: *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, *Monatshefte f. Math. u. Physik*, 38, 171–198 (1931).

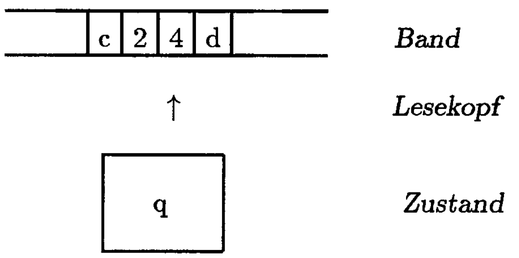
[3] A.M. Turing: *On computable numbers with an application to the Entscheidungsproblem*, *Proc. London Math. Soc.*, 2/42, 230–265 (1936). Korrektur, ibid., 43, 544–546.

1.2. Endliche Automaten und reguläre Sprachen

Deterministische endliche Automaten

Beispiel 1.2.1: Wir wollen einen Automaten konstruieren, welcher Identifiers in einer Programmiersprache erkennt. Ein Identifier ist dabei eine Folge von Buchstaben und Ziffern, welche mit einem Buchstaben beginnt.

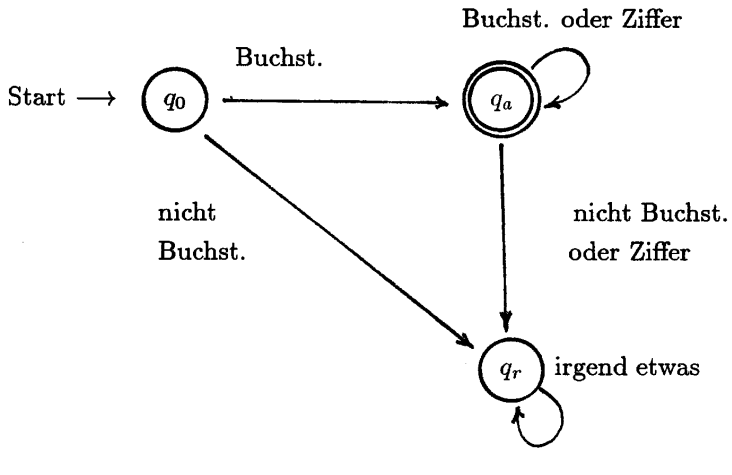
A:



Der Automat A beginnt im Zustand q_0 am Beginn der zu lesenden Folge von Symbolen. In jedem Verarbeitungsschritt liest A das Bandsymbol, auf welches der Lesekopf gerade zeigt, nimmt einen neuen Zustand an und bewegt den Lesekopf um eine Position nach rechts. Liest A im Zustand q_0 einen Buchstaben, so wird der Zustand q_a angenommen, ansonsten der Zustand q_r. Wird im Zustand q_a ein Buchstabe oder eine Ziffer gelesen, so verbleibt A im Zustand q_a. Wird im Zustand q_a etwas anderes als ein Buchstabe oder eine Ziffer gelesen, so wird der Zustand q_r angenommen. Befindet sich A im Zustand q_r, so wird dieser beibehalten, unabhängig vom gelesenen Symbol.

q_a is der sogenannte *akzeptierende* (oder *End–*) Zustand.

Der Überführungsgraph von A sieht wie folgt aus:

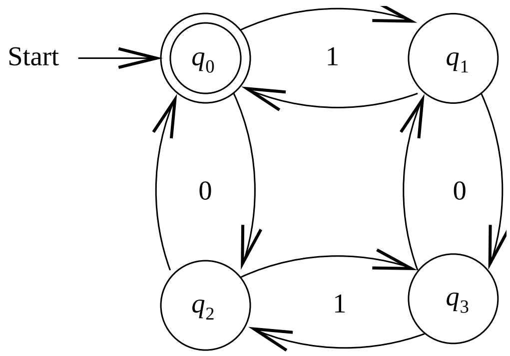


■

Beispiel 1.2.2: Ein Automat, der Folgen von 0 und 1 akzeptiert, in denen die Anzahl der 0en und 1en beide gerade sind.

Möglichkeiten:

#0	#1 gerade	#1 ungerade
gerade	q_0	q_1
ungerade	q_2	q_3



■

Wir wollen nun eine formale Definition für einen endlichen Automaten geben.

Def. 1.2.1: Ein *(deterministischer) endlicher Automat (DEA)* ist ein 5–tupel $M = (Q, \Sigma, \delta, q_0, F)$. Dabei ist

Q eine endliche Menge (von *Zuständen*),

Σ eine endliche Menge (*Eingabealphabet*),

q_0 ein Element von Q (*Startzustand*),

$F \subseteq Q$ (Menge der *Endzustände*),

δ eine Funktion von $Q \times \Sigma$ nach Q (*Überführungsfunktion*).

Im folgenden sprechen wir meist einfach von einem "endlichen Automaten", wenn wir einen DEA meinen.

Die Überführungsfunktion wird folgendermaßen auf Strings beliebiger Länge über Σ (also Elemente von Σ^*) ausgedehnt. Wir bezeichnen die erweiterte Funktion ebenfalls mit δ.

$$\delta(q, \epsilon) = q, \quad (\epsilon \text{ ist der leere String}) \qquad \text{für beliebiges } q \in Q,$$
$$\delta(q, x\alpha) = \delta(\delta(q, x), \alpha), \qquad \text{für bel. } q \in Q, \alpha \in \Sigma, x \in \Sigma^*.$$

Der String $x \in \Sigma^*$ wird von M *akzeptiert* genau dann wenn $\delta(q_0, x) \in F$.

Die von M *akzeptierte Sprache* (Teilmenge von Σ^*) ist definiert als

$$L(M) := \{x \in \Sigma^* \mid x \text{ wird von } M \text{ akzeptiert } \}.$$

Wir sagen auch M *akzeptiert* die Sprache $L(M)$. ■

Reguläre Sprachen

Im folgenden brauchen wir einige Notationen. Sei Σ ein Alphabet, also eine endliche Menge von Symbolen, und seien $L, L_1, L_2 \subseteq \Sigma^*$, also *Sprachen über* Σ.

$\overline{L} := \Sigma^* \setminus L$ (Komplement)
$L_1 \circ L_2 := \{xy \mid x \in L_1, y \in L_2\}$ (Verkettung)
$L^0 := \{\epsilon\}$
$L^i := L \circ L^{i-1}$ für $i \geq 1$
$L^* := \bigcup_{i=0}^{\infty} L^i$ (Kleene Abschluß)
$L^+ := \bigcup_{i=1}^{\infty} L^i$ (positiver Abschluß)

Def. 1.2.2: Sei Σ ein Alphabet. Ein *regulärer Ausdruck* über Σ ist entweder

$\emptyset, \epsilon, \alpha$ (für jedes Symbol $\alpha \in \Sigma$), oder

$(r + s)$, $(r \cdot s)$, (r^*), wobei r und s reguläre Ausdrücke sind.

Die Funktion L transformiert reguläre Ausdrücke über Σ in Untermengen von Σ^*:

$L(\emptyset) := \emptyset,\ L(\epsilon) := \{\epsilon\},\ L(\alpha) := \{\alpha\}$ (für jedes $\alpha \in \Sigma$),

$L(r + s) := L(r) \cup L(s)$,

$L(r \cdot s) := L(r) \circ L(s)$,

$L(r^*) := L(r)^*$.

Ist r ein regulärer Ausdruck, so heißt $L(r)$ die durch r bezeichnete *reguläre Menge* oder *reguläre Sprache*. ■

Streng genommen müßten wir in obiger Definition natürlich $L((r + s))$ schreiben anstatt $L(r + s)$, ebenso für $r \cdot s$ und r^*. Wir gestatten uns aber die abkürzende

Schreibweise, wenn kein Mißverständnis möglich ist. Dabei vereinbaren wir folgende Präzedenzregel: * vor $\cdot$ vor $+$.

Beispiel 1.2.3: Sprache der Identifiers L_I: $\Sigma = \{a, \ldots, z, 0, \ldots, 9\}$,

$$\begin{aligned} L_I =& (\{a\} \cup \cdots \cup \{z\}) \ \circ \ \Sigma^* = \\ & L(\ (a + \cdots + z) \ \cdot \ (a + \cdots + z + 0 + \cdots + 9)^* \ ). \end{aligned}$$

■

Endliche Automaten und reguläre Ausdrücke erzeugen dieselben Sprachen.

Satz 1.2.1: *Sei Σ ein Alphabet, $L \subseteq \Sigma^*$.*

(i) L ist eine reguläre Sprache

genau dann wenn

(ii) L ist die von einem endlichen Automaten akzeptierte Sprache.

Beweis: (i) $\Longrightarrow$ (ii): Sie t ein regulärer Ausdruck über Σ, der die Sprache L bezeichnet.

(a) Zunächst konstruieren wir einen neuen regulären Ausdruck t' über einem neuen Alphabet Σ', so, daß jedes Vorkommen eines $x \in \Sigma$ in t' durch einen eigenen Index gekennzeichnet ist.

Zum Beispiel: $t = a \cdot a + b^* \cdot a \cdot b^*$,
$t' = a_1 \cdot a_2 + b_1^* \cdot a_3 \cdot b_2^*$.

Sei nun $L' := L(t')$.

(b) Ausgehend von t' konstruieren wir nun folgende Mengen:
B' .... mögliche Anfangssymbole von Strings in L',
E' .... mögliche Endsymbole von Strings in L',
S' .... mögliche Paare aufeinanderfolgender Symbole in Strings in L'.

Zum Beispiel: $B' = \{a_1, b_1, a_3\}$,
$E' = \{a_2, a_3, b_2\}$,
$S' = \{(a_1, a_2), (b_1, b_1), (b_1, a_3), (a_3, b_2), (b_2, b_2)\}$.

(c) Ausgehend von B', E', S' konstruieren wir nun einen endlichen Automaten M, sodaß $L = L(M)$, auf folgende Weise:

$Q_0 := \{q_0\}$; (für irgendein Zustandssymbol q_0)
$\delta(q_0, \alpha) := \{\alpha_i \mid \alpha_i \in B'\}$ für alle $\alpha \in \Sigma$;
$k := 0$;
$Q_1 := Q_0 \cup \{\delta(q_0, \alpha) \mid \alpha \in \Sigma\}$;
while $Q_{k+1} \neq Q_k$ **do**
$\{k := k + 1$;
$\delta(q, \alpha) := \{\alpha_i \mid$ es gibt ein $\beta_j \in q$ mit $(\beta_j, \alpha_i) \in S'\}$
für alle $q \in Q_k \setminus \{q_0\}$ und $\alpha \in \Sigma$;
$Q_{k+1} := Q_k \cup \{\delta(q, \alpha) \mid q \in Q_k, \alpha \in \Sigma\}$ };
$Q := Q_k$

Diese Prozedur terminiert offensichtlich, da es nur endlich viele Symbole in Σ' gibt.

Es ist klar, daß $x \in L$ genau dann wenn M, ausgehend von q_0 und dem Eingabewort x, einen Zustand erreicht, der ein Element von E' enthält. Wenn wir also setzen

$$q \in F :\Longleftrightarrow q \cap E' \neq \emptyset \quad \text{für } q \in Q \setminus \{q_0\}, \text{ und}$$
$$q_0 \in F :\Longleftrightarrow \epsilon \in L,$$

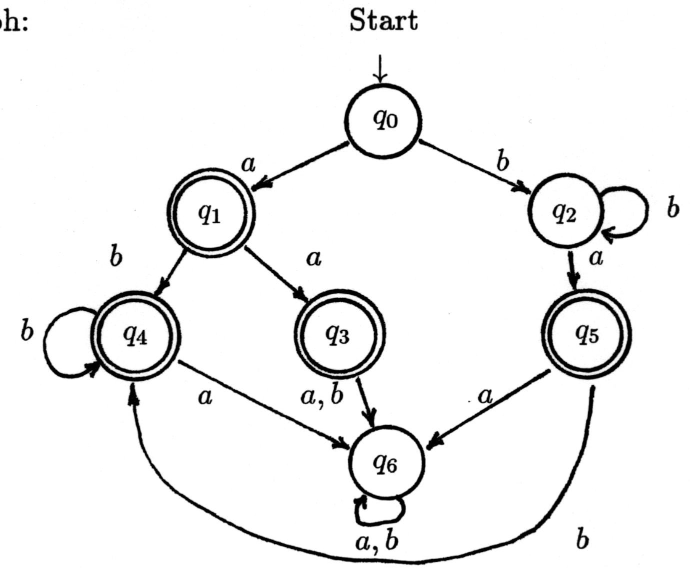
dann sind wir fertig.

Zum Beispiel: $Q_0 = \{q_0\}$,

$$\begin{aligned}
&\qquad \delta(q_0, a) = \{a_1, a_3\} =: q_1,\\
&\qquad \delta(q_0, b) = \{b_1\} =: q_2,\\
&Q_1 = \{q_0, q_1, q_2\},\\
&\qquad \delta(q_1, a) = \{a_2\} =: q_3,\\
&\qquad \delta(q_1, b) = \{b_2\} =: q_4,\\
&\qquad \delta(q_2, a) = \{a_3\} =: q_5,\\
&\qquad \delta(q_2, b) = \{b_1\} =: q_2,\\
&Q_2 = \{q_0, \ldots, q_5\},\\
&\qquad \delta(q_3, a) = \emptyset =: q_6,\\
&\qquad \delta(q_3, b) = \delta(q_4, a) = \delta(q_5, a) = \emptyset = q_6,\\
&\qquad \delta(q_4, b) = \delta(q_5, b) = \{b_2\} = q_4,\\
&Q_3 = \{q_0, \ldots, q_6\},\\
&\qquad \delta(q_6, a) = \delta(q_6, b) = \emptyset = q_6,\\
&Q_4 = Q_3.
\end{aligned}$$

Also $Q = Q_3 = \{q_0, \ldots, q_6\}$, und $F = \{q_1, a_3, q_4, q_5\}$.

Überführungsgraph:



(ii) $\Longrightarrow$ (i): Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein endlicher Automat mit $L = L(M)$. Für beliebige Zustände $q, p \in Q$ betrachten wir die Mengen

$$R_{q,p} := \{x \in \Sigma^* \mid \delta(q, x) = p\}.$$

Es gilt

$$L = \bigcup_{p \in F} R_{q_0,p}. \tag{1}$$

Nun sei $Q = \{q_0, q_1, \ldots, q_r\}$. Für beliebige $q, p \in Q$ und $1 \leq j \leq r+1$ definieren wir:

$$R^j_{q,p} := \{x \in R_{q,p} \mid \delta(q, y) \in \{q_0, \ldots, q_{j-1}\} \text{ für jeden Prefix } y \text{ von } x\} \tag{2}$$

und

$$R^0_{q,p} := \begin{cases} \{\alpha \in \Sigma \mid \delta(q, \alpha) = p\}, & \text{falls } q \neq p, \\ \{\alpha \in \Sigma \mid \delta(q, \alpha) = p\} \cup \{\epsilon\}, & \text{falls } q = p. \end{cases} \tag{3}$$

Dann gilt

$$R_{q,p} = R^{r+1}_{q,p}. \tag{4}$$

Für alle q, p ist $R^0_{q,p}$ eine reguläre Menge, da es die Vereinigung endlich vieler Mengen $\{a\}, a \in \Sigma$, oder $\{\epsilon\}$ ist. Weiters gilt

$$R^{j+1}_{q,p} = R^j_{q,p} \ \cup \ R^j_{q,q_j} \circ (R^j_{q_j,q_j})^* \circ R^j_{q_j,p}. \tag{5}$$

Aus (5) sehen wir, daß $R^j_{q,p}$ $(1 \leq j \leq r+1)$ reguläre Mengen sind. Somit ist wegen (4) und (1) auch L regulär. Tatsächlich haben wir sogar einen regulären Ausdruck konstruiert, der L bezeichnet. ∎

Beispiel 1.2.4: Wir betrachten den im ersten Teil des Satzes konstruierten Automaten. Für diesen gilt

$$L = R_{q_0,q_1} \cup R_{q_0,q_3} \cup R_{q_0,q_4} \cup R_{q_0,q_5}.$$

$$\begin{aligned}
R_{q_0,q_1} =& R^7_{q_0,q_1} \\
& R^0_{q_0,q_1} = \{a\}, \\
& R^1_{q_0,q_1} = \{a\} \ \cup \ \underbrace{R^0_{q_0,q_1}}_{\{a\}} \circ \underbrace{(R^0_{q_1,q_1})^*}_{\{\epsilon\}^*} \circ \underbrace{R^0_{q_1,q_1}}_{\{\epsilon\}} = \{a\}, \\
& R^2_{q_0,q_1} = \cdots = R^7_{q_0,q_1} = \{a\}
\end{aligned}$$

etc. ■

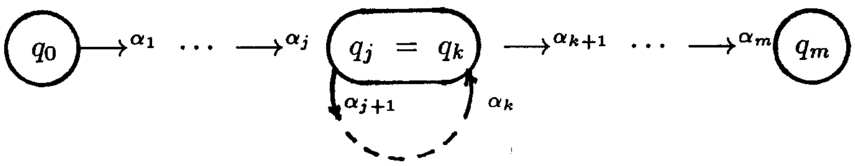
Der folgende Satz eignet sich oft sehr gut dazu zu entscheiden, ob eine vorliegende Sprache regulär ist oder nicht. Vereinfacht gesagt ist eine Sprache nur dann regulär, wenn genügend lange Wörter in ihr aufgebläht werden können ohne die Sprache zu verlassen. Daher kommt auch der Name für diesen Satz.

Satz 1.2.2: (Pumping Lemma) *Sei L eine reguläre Sprache. Dann gibt es eine Konstante $n (\in \mathbb{N})$, sodaß jedes Wort $z \in L$ der Länge mindestens n ($\mid z \mid \geq n$) auf solche Weise zerlegt werden kann in $z = uvw$, daß $\mid uv \mid \leq n$, $\mid v \mid \geq 1$, und für alle $i \geq 0$ auch $uv^iw \in L$. Die Konstante n ist dabei nicht größer als die Anzahl der Zustände des kleinsten Automaten, der L akzeptiert.*

Beweis: Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein endlicher Automat, der L akzeptiert. Sei $n = \mid Q \mid$. Wir betrachten nun ein Wort z, sodaß

$$z = \alpha_1 \cdots \alpha_m \in L \quad \text{und} \quad \mid z \mid = m \geq n.$$

Wenn wir mit q_i denjenigen Zustand bezeichnen, welcher nach Abarbeitung von $\alpha_1 \cdots \alpha_i$ angenommen wird, so müssen offensichtlich zwei solche Zustände identisch sein. Also für gewisse j, k ($0 \leq j < k \leq n$),



z kann also zerlegt werden in $z = uvw$, wobei

$$u = \alpha_1 \cdots \alpha_j, \quad v = \alpha_{j+1} \cdots \alpha_k, \quad w = \alpha_{k+1} \cdots \alpha_m.$$ ■

Beispiel 1.2.5: $L = \{a^m b^m \mid m \in \mathbb{N}\}$ ist keine reguläre Sprache.

Angenommen L wäre regulär. Sei n die Konstante im Pumping Lemma. Wir betrachten das Wort

$$z = a^n b^n,$$

das natürlich in L enthalten ist. Wegen des Pumping Lemmas muß z zerlegbar sein in

$$a^n b^n = uvw,$$

wobei $|uv| \leq n$, $|v| \geq 1$, und $uv^i w \in L$ für jedes i. Somit müßte gelten

$$\underbrace{a^{n_1}}_{u}\ \underbrace{a^{n_2}}_{v}\ \underbrace{a^{n_3} b^n}_{w}, \qquad \text{wobei } n = n_1 + n_2 + n_3,\ n_2 \geq 1.$$

Aber $a^{n_1}(a^{n_2})^2 a^{n_3} b^n$ ist nicht in L. Somit kann L nicht regulär sein. ■

Beispiel 1.2.6: $L = \{0^{i^2} \mid i \in \mathbb{N}\}$ ist keine reguläre Sprache.

Angenommen L wäre regulär. Sei n die Konstante im Pumping Lemma. Wir betrachten das Wort

$$z = 0^{n^2},$$

das natürlich in L enthalten ist. Wegen des Pumping Lemmas muß z zerlegbar sein in

$$0^{n^2} = uvw,$$

wobei $|uv| \leq n$, $|v| \geq 1$, und $uv^i w \in L$ für jedes i. Das ist aber für $i = 2$, also $uv^2 w$, schon nicht der Fall. $n^2 < |uv^2 w| \leq n^2 + n < (n+1)^2$. Somit ist $uv^2 w \notin L$. ■

Nichtdeterministische endliche Automaten

Man kann sich nun aber auch endliche Automaten vorstellen, die bei gegebenem Zustand und Eingabesymbol nicht nun einen einzig möglichen Nachfolgezustand haben, sondern aus verschiedenen (oder auch gar keinen) Möglichkeiten für Nachfolgerzustände wählen können.

Wie soll die Semantik eines solchen "nichtdeterministischen endlichen Automaten" aussehen? Wenn im Zustand p der Automat das Symbol a liest und etwa $\{q_1, q_2\}$ für diesen Fall als mögliche Nachfolgezustände definiert sind, so kann der Automat entweder in den Zustand q_1 oder in den Zustand q_2 übergehen. Der Automat akzeptiert ein Wort $w \in \Sigma^*$, wenn es einen möglichen Berechnungspfad gibt, sodaß der Automat von einem Anfangszustand und Wort w auf dem Band zu einem Endzustand gelangen kann.

Def. 1.2.3: Ein *nichtdeterministischer endlicher Automat (NEA)* ist ein 5–tupel $M = (Q, \Sigma, \delta, S, F)$. Dabei ist

Q eine endliche Menge (von *Zuständen*),

Σ eine endliche Menge (*Eingabealphabet*),

$S \subseteq Q$ (Menge der *Anfangszustände*),

$F \subseteq Q$ (Menge der *Endzustände*),

δ eine Funktion von $Q \times \Sigma$ nach $\mathcal{P}(Q)$, der Potenzmenge von Q (*Überführungsfunktion*).

Die Überführungsfunktion wird folgendermaßen ausgedehnt zu einer Funktion von $Q \times \Sigma^*$ nach $\mathcal{P}(Q)$. Wir bezeichnen die erweiterte Funktion ebenfalls mit δ.

$\delta(q, \epsilon) = \{q\}$, für bel. $q \in Q$,

$\delta(q, x\alpha) = \{p \mid p \in \delta(r, \alpha) \text{ für ein } r \in \delta(q, x)\}$, für bel. $q \in Q, x \in \Sigma^*, \alpha \in \Sigma$.

Weiters dehnen wir δ aus zu einer Funktion von $\mathcal{P}(Q) \times \Sigma^*$ nach $\mathcal{P}(Q)$:

$\delta(P, x) = \bigcup_{p \in P} \delta(p, x)$, für bel. $P \subseteq Q$, $x \in \Sigma^*$.

Der String $x \in \Sigma^*$ wird von M *akzeptiert* genau dann wenn $\delta(S, x) \cap F \neq \emptyset$.

Die von M *akzeptierte Sprache* (Teilmenge von Σ^*) ist definiert als

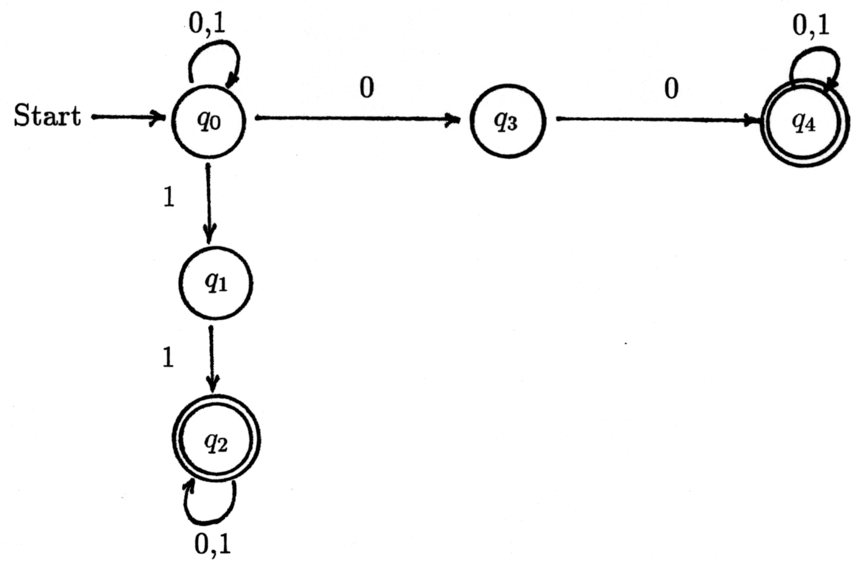
$$L(M) := \{x \in \Sigma^* \mid x \text{ wird von } M \text{ akzeptiert }\}.$$

Wir sagen auch M *akzeptiert* die Sprache $L(M)$. ∎

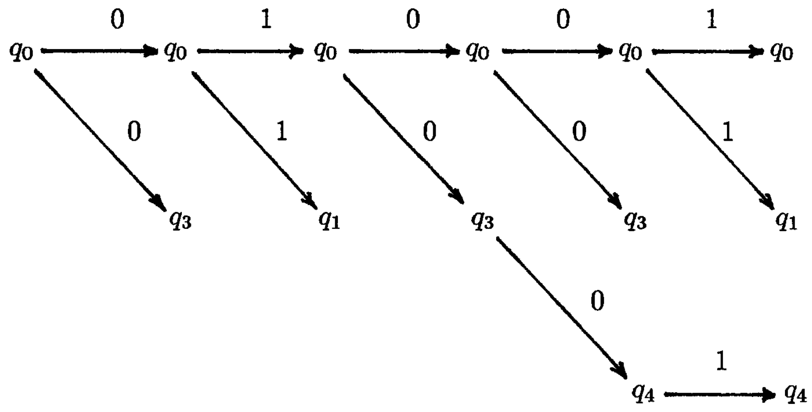
Beispiel 1.2.7: Wir betrachten folgenden NEA M, wobei $Q = \{q_0, q_1, q_2, q_3, q_4\}, \Sigma = \{0, 1\}, S = \{q_0\}, F = \{q_2, q_4\}$, und die Überführungsfunktion δ durch folgende Tabelle gegeben ist:

	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$

Der Überführungsgraph dieses NEA M sieht folgendermaßen aus:



Wenn der Automat die Möglichkeit hat, wie im Zustand q_0 bei Eingabe 0, verschiedene Nachfolgezustände anzunehmen, so können wir uns vorstellen, daß entsprechend viele Kopien des Automaten hergestellt werden, deren jede eine dieser Möglichkeiten realisiert. Diese Form der Abarbeitung eines Eingabestrings führt zu einem Berechnungsbaum. Wir stellen den Berechnungsbaum für den NEA M bei Eingabe 01001 dar:



Die von M akzeptierte Sprache $L(M)$ besteht aus all denjenigen Wörtern, welche 00 oder 11 enthalten. ■

Es ist offensichtlich, daß jeder DEA aufgefaßt werden kann als ein NEA, bei dem eben die Menge der möglichen Nachfolgezustände jeweils eine einelementige Menge ist. Somit ist klar, daß jede Sprache L, welche von einem DEA akzeptiert wird, auch von einem NEA akzeptiert werden kann. Interessanterweise gilt auch die Umkehrung.

Satz 1.2.3: *Sei M ein NEA und sei $L = L(M)$. Dann gibt es einen DEA M' mit $L(M') = L$.*

Beweis: Sei $M = (Q, \Sigma, \delta, S, F)$ der gegebene NEA. Wie konstruieren einen DEA M', der die gleiche Sprache akzeptiert, indem wir jede Teilmenge von Q als Zustand von M' auffassen. Alles übrige ergibt sich dann fast von selbst. Wir setzen also

$$M' = (Q', \Sigma, \delta', q'_0, F'),$$

wobei

$$\begin{aligned} Q' &= \mathcal{P}(Q) \qquad \text{(Potenzmenge von } Q\text{)}, \\ \delta'(q', \alpha) &= \bigcup_{p \in q'} \delta(p, \alpha) \ (= \delta(q', \alpha)\,), \quad \text{für } q' \in Q', \\ q'_0 &= S, \\ F' &= \{q' \in Q' \mid q' \cap F \neq \emptyset\}. \end{aligned}$$

Nun sieht man leicht, daß für ein beliebiges Wort $x = \alpha_1 \ldots \alpha_n \in \Sigma^*$ gilt:

$x \in L(M)$

$\Longleftrightarrow$

$\delta(S, x) \cap F \neq \emptyset$

$\Longleftrightarrow$

es gibt eine Folge von Teilmengen $S = Q_0, \ldots, Q_n$ von Q, bzw. Zuständen $q'_0 = Q_0, \ldots, q'_n = Q_n$ von M', sodaß $\delta'(q'_0, \alpha_1) = q'_1, \ldots, \delta'(q_{n-1}, \alpha_n) = q'_n$, und $q'_n \cap F \neq \emptyset$

$\Longleftrightarrow$

$\delta'(q'_0, x) \in F'$

$\Longleftrightarrow$

$x \in L(M')$. ∎

Die Erweiterung des Begriffs des endlichen Automaten zu einem nichtdeterministischen endlichen Automaten bringt also keine Potenzsteigerung der betrachteten Klasse von Automaten. Diese Beobachtung werden wir auch später wieder machen, wenn wir uns mit Turing–Maschinen befassen.

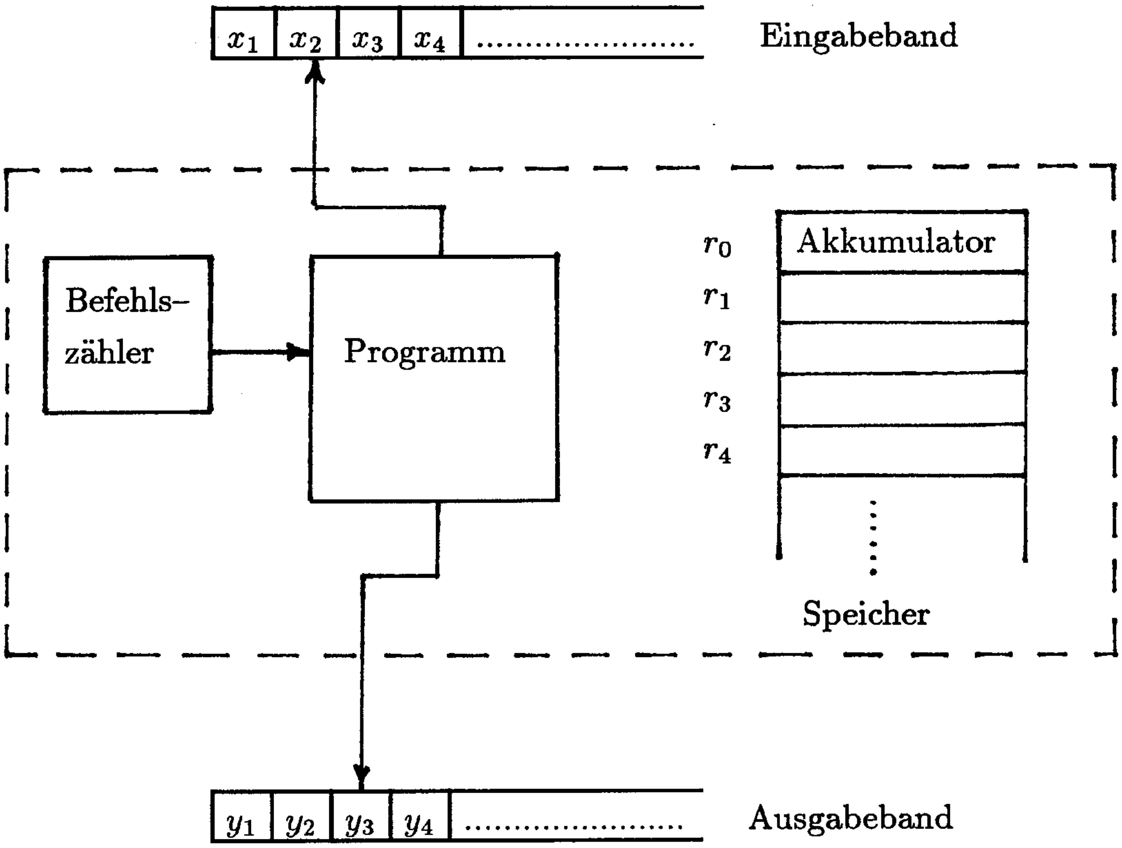
Neben den besprochenen Beschreibungsmöglichkeiten für reguläre Sprachen (von DEA akzeptiert, von NEA akzeptiert, durch einen regulären Ausdruck erzeugt) gibt es noch einige andere. So könnte man etwa auch reguläre Grammatiken betrachten. Diese führen wiederum zur selben Menge der regulären Sprachen.

Endliche Automaten kann man natürlich auch mit einer Ausgabefunktion versehen und auf diese Weise Funktionen berechnen lassen.

Wir wollen auf diese Variationen aber nicht mehr näher eingehen sondern verweisen auf die angefügte Literaturliste, insbesondere [HU] und [LP].

1.3. RAM und RASP

Definition 1.3.1: Eine *RAM (random access machine)* besteht aus einem *Eingabeband*, einem *Ausgabeband*, einem *Befehlszähler*, einem *Programm* und einem (unendlichen) *Speicher*.



Das Eingabeband ist eine unendliche Folge von *Feldern* und das Ausgabeband ist eine unendliche Folge von Feldern, von denen jedes eine ganze Zahl beinhalten kann. Der Speicher ist eine unendliche Folge von *Registern*, deren jedes eine ganze Zahl aufnehmen kann. Das Programm ist eine endliche Folge von (eventuell markierten) *Instruktionen*. Jede Instruktion besteht aus zwei Teilen, einem *Operationskode* und einer *Adresse* (die entweder ein *Operand* oder eine *Marke* ist).

Liste der RAM–Instruktionen:

Operationskode	Adresse	Operationskode	Adresse
1. `LOAD`	Operand	7. `READ`	Operand
2. `STORE`	Operand	8. `WRITE`	Operand
3. `ADD`	Operand	9. `JUMP`	Marke
4. `SUB`	Operand	10. `JGTZ`	Marke
5. `MULT`	Operand	11. `JZERO`	Marke
6. `DIV`	Operand	12. `HALT`	

Ein Operand kann wie folgt beschaffen sein:

1. `=i`, die Bedeutung ist die ganze Zahl i selbst,
2. `i`, wobei i eine nichtnegative ganze Zahl ist. Die Bedeutung ist der Inhalt des Registers i,
3. `*i`, die Bedeutung ist der Inhalt von Register j, wobei j die ganze Zahl in Register i ist (indirekte Adressierung). Ist $j < 0$, so bleibt die Maschine stehen.

Um die *Bedeutung* einer RAM R zu definieren, ziehen wir die *Speicherfunktion* $c : \mathbb{N}_0 \to \mathbb{Z}$ heran. $c(i)$ ist der Inhalt des Speicherregisters r_i.

Am Anfang der *Berechnung* ist $c(i) = 0$ für alle $i \geq 0$, der Befehlszähler ist auf 1 gesetzt, und jedes Feld des Ausgabebandes enthält Blank. Bis auf endlich viele Ausnahmen sind alle Felder des Eingabebandes leer. Der Lesekopf zeigt auf das erste Feld des Eingabebandes und der Schreibkopf zeigt auf das erste Feld des Ausgabebandes. Das Feld, auf welches der Lese– bzw. Schreibkopf gerade zeigt, heißt das *aktuelle* Eingabe– bzw. Ausgabefeld. In jedem Schritt der Berechnung wird diejenige Instruktion ausgeführt, auf die der Befehlszähler gerade zeigt, also die k–te Instruktion, wenn k der Inhalt des Befehlszählers ist. Nach Ausführung der k–ten Instruktion von R wird der Befehlszähler automatisch auf $k+1$ gesetzt, wenn die k–te Instruktion nicht eine der Anweisungen `JUMP`, `HALT`, `JGTZ` oder `JZERO` war. Die Berechnung endet, wenn die Instruktion `HALT` abgearbeitet wird, oder zu einer Marke gesprungen wird, die im Programm nicht vorkommt.

Um die Bedeutung einer Instruktion anzugeben, definieren wir $v(a)$, den *Wert* des Operanden a, wie folgt:

$$v(=\mathtt{i}) = i, \quad v(\mathtt{i}) = c(i), \quad v(*\mathtt{i}) = c(c(i)).$$

Die folgende Tabelle gibt die Bedeutung der einzelnen Instruktionen an. Instruktionen, deren Bedeutung in der Tabelle nicht definiert ist, wie etwa `STORE =i`, haben dieselbe Bedeutung wie `HALT`. Ebenso beendet eine Division durch 0 die Berechnung.

Instruktion	Bedeutung
1. `LOAD a`	$c(0) \leftarrow v(a)$
2. `STORE i`	$c(i) \leftarrow c(0)$
`STORE *i`	$c(c(i)) \leftarrow c(0)$
3. `ADD a`	$c(0) \leftarrow c(0) + v(a)$
4. `SUB a`	$c(0) \leftarrow c(0) - v(a)$
5. `MULT a`	$c(0) \leftarrow c(0) \cdot v(a)$
6. `DIV a`	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$
7. `READ i`	$c(i) \leftarrow$ Symbol im aktuellen Eingabefeld. Der Lesekopf wird um eine Position nach rechts gerückt.
`READ *i`	$c(c(i)) \leftarrow$ Symbol im aktuellen Eingabefeld. Der Lesekopf wird um eine Position nach rechts gerückt.

8. `WRITE a`	$v(a)$ wird in das aktuelle Ausgabefeld geschrieben. Der Schreibkopf wird um eine Position nach rechts gerückt.
9. `JUMP b`	Der Befehlszähler wird auf die Nummer der Instruktion mit der Marke b gesetzt.
10. `JGTZ b`	Falls $c(0) > 0$, dann wird der Befehlszähler auf die Nummer der Instruktion mit der Marke b gesetzt. Andernfalls wird der Befehlszähler um 1 erhöht.
11. `JZERO b`	Falls $c(0) = 0$, dann wird der Befehlszähler auf die Nummer der Instruktion mit der Marke b besetzt. Andernfalls wird der Befehlszähler um 1 erhöht.
12. `HALT`	Die Berechnung endet.

Die *Bedeutung* der RAM R ist die durch R definierte Abbildung von Eingabebändern zu Ausgabebändern. Da die Berechnung von R nicht auf allen Eingabebändern terminieren muß, ist diese Abbildung nur eine partielle Abbildung. Diese Abbildung kann nun in verschiedener Weise interpretiert werden, etwa als Funktion oder als Sprache.

Die RAM R lese immer genau n Zahlen $x_1, \ldots, x_n$ vom Eingabeband, und schreibe genau eine Zahl y auf das Ausgabeband oder terminiere nicht. Damit ist implizit die (partielle) Funktion

$$f(x_1, \ldots, x_n) = \begin{cases} y & \text{falls } R \text{ auf } x_1, \ldots, x_n \text{ terminiert und } y \text{ schreibt} \\ \perp & \text{falls } R \text{ auf } x_1, \ldots, x_n \text{ nicht terminiert} \end{cases}$$

erklärt. Wir sagen R *berechnet* die Funktion f.

Man kann R auch auffassen als Akzeptor einer Sprache *) über einem Alphabet $\{\alpha_1, \ldots, \alpha_k\}$. Das Symbol α_i kann kodiert werden als die ganze Zahl i. Ein Wort wird so auf das Eingabeband geschrieben, daß die Kodes für die einzelnen Symbole nacheinander in die Eingabefelder geschrieben werden und als Abschluß in das nächste Feld eine 0 geschrieben wird. Man sagt nun das Wort s wird von R *akzeptiert*, wenn R das Wort s und das Markierungssymbol 0 vom Eingabeband liest, eine 1 in das erste Feld des Ausgabebandes schreibt, und hält. Die *von R akzeptierte Sprache* ist die Menge aller Wörter, die von R akzeptiert werden. ■

Beispiel 1.3.1: Wir betrachten die Funktion $f(n)$, gegeben als

$$f(n) = \begin{cases} n^n & \text{falls } n \text{ positiv ist} \\ 0 & \text{sonst.} \end{cases}$$

*) Vgl. Sektion 1.2. Ein Alphabet Σ ist eine endliche Menge von Symbolen. Ein Wort w über Σ ist eine endliche Folge von Symbolen aus Σ. Eine Sprache L über Σ ist eine Teilmenge von Σ^*, der Menge aller Wörter über Σ.

Ein Programm zur Berechnung von $f(n)$ (in einer PASCAL–ähnlichen Sprache) könnte wie folgt aussehen:

begin
  read $r1$;
  if $r1 \leq 0$ **then write** 0
  else
    begin
      $r2 \leftarrow r1$;
      $r3 \leftarrow r1 - 1$
      while $r3 > 0$ **do**
        begin
          $r2 \leftarrow r2 * r1$;
          $r3 \leftarrow r3 - 1$
        end;
      write $r2$
    end
end

Ein entsprechendes RAM Programm ist

RAM Programm:			entsprechende PASCAL Statements:
	`READ`	`1`	**read** $r1$
	`LOAD`	`1`	
	`JGTZ`	`pos`	**if** $r1 \leq 0$ **then write** 0
	`WRITE`	`=0`	
	`JUMP`	`endif`	
`pos:`	`LOAD`	`1`	$r2 \leftarrow r1$
	`STORE`	`2`	
	`LOAD`	`1`	
	`SUB`	`=1`	$r3 \leftarrow r1 - 1$
	`STORE`	`3`	
`while:`	`LOAD`	`3`	
	`JGTZ`	`continue`	**while** $r3 > 0$ **do**
	`JUMP`	`endwhile`	

```
continue:   LOAD    2
            MULT    1           r2 ← r2 * r1
            STORE   2
            LOAD    3
            SUB     =1          r3 ← r3 - 1
            STORE   3
            JUMP    while
endwhile:   WRITE   2           write r2
endif:      HALT
```

■

Beispiel 1.3.2: Wir erzeugen ein RAM Programm, welches die Sprache über dem Alphabet $\{1, 2\}$ akzeptiert, die aus allen Wörtern mit gleich vielen 1en wie 2en besteht. Das Programm liest jedes Eingabesymbol in Register 1 ein. In Register 2 wird jeweils die Differenz der Anzahl der bereits gelesenen 1en und 2en gespeichert. Sobald das Programm zum Endsymbol 0 kommt, prüft es, ob die Differenz 0 ist. Wenn ja, so wird 1 ausgedruckt und das Programm hält. Wir nehmen an, daß 0, 1 und 2 die einzigen möglichen Eingabesymbole sind.

Ein Pidgin–PASCAL Programm könnte wie folgt aussehen:

```
begin
    d ← 0;
    read x;
    while x ≠ 0 do
        begin
            if x ≠ 1 then d ← d - 1 else d ← d + 1;
            read x
        end;
    if d = 0 then write 1
end
```

Ein entsprechendes RAM Progamm ist

```
            LOAD    =0          d ← 0
            STORE   2
            READ    1           read x
while:      LOAD    1           while x ≠ 0 do
            JZERO   endwhile
            LOAD    1
            SUB     =1          if x ≠ 1
            JZERO   one
            LOAD    2
            SUB     =1          then d ← d - 1
            STORE   2
            JUMP    endif
one:        LOAD    2
            ADD     =1          else d ← d + 1
```

```
              STORE   2
endif:        READ    1           read x
              JUMP    while
endwhile:     LOAD    2
              JZERO   output      if d = 0 then write 1
              HALT
output:       WRITE   =1
              HALT
```

■

Das Programm einer RAM ist nicht im Speicher abgelegt, es kann sich also nicht selbst modifizieren. Wir wollen nun ein Maschinenmodell betrachten, das ganz gleich ist wie eine RAM, dessen Programm aber im Speicher abgelegt ist.

Definition 1.3.2: Die Instruktionen einer *RASP (random access stored program* machine) sind identisch mit den Instruktionen einer RAM, indirekte Adressierung ist jedoch nicht gestattet (eine RASP kann indirekte Adressierung simulieren durch Modifikation der Instruktionen während der Ausführung des Programms). Die Struktur einer RASP ist auch gleich wie die einer RAM, nur daß das Programm im Speicher abgelegt ist. Jede RASP Instruktion belegt zwei aufeinanderfolgende Register. Im ersten Register steht eine Kodierung des Operationskodes, im zweiten Register eine Adresse. Ist die Adresse von der Form `=i`, dann kodiert der Inhalt des ersten Registers auch den Umstand, daß der Operand ein Literal ist, und das zweite Register enthält i. Die Instruktionen werden als Zahlen kodiert, etwa auf folgende Weise:

Instruktion:	Kode:	Instruktion:	Kode:
`LOAD i`	1	`DIV i`	10
`LOAD =i`	2	`DIV =i`	11
`STORE i`	3	`READ i`	12
`ADD i`	4	`WRITE i`	13
`ADD =i`	5	`WRITE =i`	14
`SUB i`	6	`JUMP i`	15
`SUB =i`	7	`JGTZ i`	16
`MULT i`	8	`JZERO i`	17
`MULT =i`	9	`HALT`	18

So würde etwa die Instruktion `LOAD =32` abgespeichert als 2 in einem Register und 32 im nächsten Register.

Die Abarbeitung eines RASP Programms erfolgt im wesentlichen gleich wie die Abarbeitung eine RAM Programms, mit den folgenden geringfügigen Änderungen: zu Beginn der Berechnung ist das Programm in zusammenhängenden Speicherregistern abgelegt, und der Befehlszähler ist auf die erste Instruktion gesetzt. Die übrigen Register enthalten alle 0. Nach der Ausführung jeder Instruktion wird der Befehlszähler um 2 erhöht, außer nach einer Instruktion der Form `JUMP i`, `JGTZ i` (wenn der Inhalt des Akkumulators positiv ist) oder `JZERO i` (wenn der Inhalt des Akkumulators 0 ist). In diesen Fällen wird der Befehlszähler zu i gesetzt. ∎

Jede Funktion (Sprache), welche von einem RAM Programm berechnet (akzeptiert) werden kann, kann auch von einem RASP Programm berechnet (akzeptiert) werden und umgekehrt; das wird im folgenden Satz ausgesprochen.

Satz 1.3.1: *Zu jedem RAM Programm gibt es ein äquivalentes RASP Programm und umgekehrt.*

Beweis: (a) Zunächst zeigen wir, wie ein RAM Programm P durch ein RASP Programm P' simuliert werden kann. Register 1 der RASP wird zur Zwischenspeicherung des RAM Akkumulators benutzt. Das Programm P' belegt die nächsten $r-1$ Register der RASP. Die Konstante r hängt ab vom RAM Programm P. Der Inhalt des RAM Registers i, $i \geq 1$, wird im RASP Register $r+i$ abgespeichert, sodaß alle Speicherzugriffe in P' um r gegenüber P verschoben sind.
Jede RAM Instruktion, die keine indirekte Adressierung benutzt, wird in die identische RASP Instruktion (mit entsprechend verschobenen Speicherzugriffen) übersetzt. Jede RAM Instruktion mit indirekter Adressierung wird übersetzt in eine Folge von sechs RASP Instruktionen; dabei wird die indirekte Adressierung durch Instruktionsmodifikation simuliert. Als Beispiel dafür betrachten wir die RAM Instruktion `SUB *i`, $i \geq 1$. Diese RAM Instruktion wird in eine Folge von RASP Instruktionen übersetzt, die

1. den Inhalt des Akkumulators in Register 1 zwischenspeichert,
2. den Inhalt von Register $r+i$ in den Akkumulator lädt (Register $r+i$ der RASP entspricht dem Register i der RAM),
3. r zum Akkumulator addiert,
4. den Inhalt des Akkumulators in das Adressenfeld einer `SUB` Instruktion abspeichert,
5. den Inhalt von Register 1 wieder in den Akkumulator zurücklädt, und schließlich
6. die in Schritt 4 erzeugte `SUB` Instruktion benutzt, um die Subtraktion auszuführen. So wird, unter der Annahme, daß die Folge der RASP Instruktionen in Register 100 beginnt, die RAM Instruktion `SUB *i` wie folgt simuliert:

Register	Inhalt	Bedeutung
100	3	`STORE 1`
101	1	
102	1	`LOAD` $r+i$
103	$r+i$	
104	5	`ADD =`r
105	r	
106	3	`STORE 111`
107	111	
108	1	`LOAD 1`
109	1	
110	6	`SUB` b, wobei b der Inhalt des RAM Registers i ist.
111	–	

Enthält P n Instruktionen mit m indirekten Adressierungen, so enthält P' $(n-m)+6m$ Instruktionen. Der Verschiebungsfaktor r ist also $2n+10m+1$.

(b) Zu einem gegebenen RASP Programm P' konstruieren wir ein äquivalentes RAM Programm P. Einige RAM Register werden für spezielle Zwecke benutzt:

Register 1:	für indirekte Adressierung,
Register 2:	der Befehlszähler der RASP,
Register 3:	der Akkumulator der RASP.

Register i, $i \geq 1$, der RASP wird in Register $i+3$ der RAM gespeichert.

Die RAM beginnt mit dem RASP Programm im Speicher ab Register 4. Register 2, der RASP Befehlszähler, enthält 4. Die Register 1 und 3 enthalten 0. Das RAM Programm ist eine Simulationsschleife, welche damit beginnt eine RASP Instruktion zu lesen (mit einer `LOAD *2` Instruktion), diese dekodiert und zu einer von 18 Instruktionsfolgen verzweigt, deren jede einen Typ von RASP Instruktionen simuliert. Als Beispiel betrachten wir die RAM Instruktionsfolge zur Simulation der RASP Instruktion 6, also `SUB i`:

`LOAD 2`
`ADD =1`
`STORE 2`
	erhöhe den Befehlszähler um 1, sodaß er nun auf das Register zeigt, welches den Operanden i der `SUB i` Instruktion enthält;

`LOAD *2`
`ADD =3`
`STORE 1`
	lade i in den Akkumulator, addiere 3 und speichere das Ergebnis in Register 1;

`LOAD 3`
`SUB *1`
`STORE 3`
	lade den Inhalt des RASP Akkumulators aus Register 3, subtrahiere den Inhalt des Registers $i+3$ und lege das Ergebnis in Register 3 ab;

`LOAD 2`
`ADD =1`
`STORE 2`
	erhöhe den Befehlszähler um 1, sodaß er nun auf die nächste RASP Instruktion zeigt;

`JUMP a`	verzweige zum Anfang der Simulationsschleife (hier mit “`a`” bezeichnet).

Zu diesem Programmstück wird verzweigt, wenn $c(c(2)) = 6$, der Befehlszähler also auf ein Register mit Inhalt 6, dem Kode von `SUB`, zeigt. ■

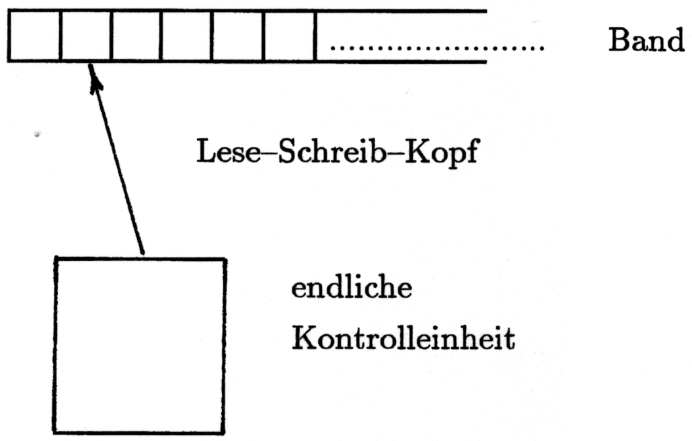
Beispiel 1.3.3: Es ist nicht schwer zu zeigen, daß die Sprachen aus den Beispielen 1.2.5 und 1.2.6 von einer RAM akzeptiert werden können. ■

Wie man auch unschwer sieht, kann jeder endliche Automat durch eine RAM simuliert werden. Wegen Beispiel 1.3.3 wissen wir also, daß das Konzept der RAM stärker ist als dasjenige endlicher Automaten, da mehr Sprachen akzeptiert werden können.

1.4. Turing–Maschinen und rekursiv aufzählbare Sprachen

Um zu beweisen, daß eine bestimmte Funktion "berechenbar" ist, oder ein gewisses Minimum an Zeit zur Berechnung benötigt (untere Komplexitätsschranke), brauchen wir ein Maschinenmodell, das zwar so allgemein aber einfacher als eine RAM oder RASP ist. Dieses Modell ist die Turing–Maschine.

Definition 1.4.1: Eine *Turing–Maschine* M können wir uns folgendermaßen vorstellen:



Sie besteht aus einer *endlichen Kontrolleinheit*, welche im Inneren eine endliche Anzahl von Zuständen aus einer *Zustandsmenge* $Q = \{q_0, \ldots, q_n\}$ annehmen kann, einem in eine Richtung unendlichen *Band*, welches in Zellen eingeteilt ist, deren jede ein Symbol aus einem endlichen *Alphabet von Bandsymbolen* $\Gamma = \{\gamma_0, \ldots, \gamma_m\}$ speichern kann — γ_0 wird dabei als das Blank Symbol $\sqcup$ aufgefaßt und eine Untermenge Σ von $\Gamma \setminus \{\sqcup\}$ ist das *Eingabealphabet* — , sowie aus einem *Lese–Schreib–Kopf*, welcher immer auf eine Bandzelle zeigt.

Wie funktioniert nun die Turing–Maschine M? Wenn die Maschine zu arbeiten beginnt, ist die endliche Kontrolleinheit im Anfangszustand q_0, und bis auf endlich viele Zellen am linken Ende des Bandes sind alle Bandzellen mit $\sqcup$'s belegt. In diskreten Zeitschritten, abhängig vom Zustand und vom Bandsymbol, auf welches der LS–Kopf gerade zeigt (Symbol in der *Arbeitszelle*), geschieht folgendes:

- M nimmt einen neuen Zustand an,
- schreibt ein Bandsymbol in die Arbeitszelle, und
- bewegt den LS–Kopf um eine Position nach rechts (R) oder nach links (L) oder läßt ihn stationär (S).

Erreicht dabei M einen Zustand aus einer Menge von *Endzuständen* F (Teilmenge der Zustandsmenge), so terminiert die Berechnung erfolgreich, M *akzeptiert* die Eingabe. Ansonsten fährt M in der Berechnung so lange fort, als zum jeweiligen Zustand und Symbol in der Arbeitszelle ein Nachfolgerzustand erklärt ist. Eine Berechnung, welche nicht erfolgreich terminiert, kann also auch unendliche Länge haben. ■

Beispiel 1.4.1: Als Beispiel betrachten wir die Turing–Maschine M_1, welche erkennt, ob eine endliche Folge x über $\{0, 1\}$ die Form $0^n 1^n$ hat, für eine natürliche Zahl n. Bei Anfang der Berechnung steht die Folge x in den ersten Zellen des Bandes von M_1, gefolgt von einer unendlichen Folge von $\sqcup$'s. M_1 führt folgende Berechnungsschleife aus:

> M_1 ersetzt die am weitesten links stehende 0 durch X, bewegt den LS–Kopf nach rechts bis zur am weitesten links stehenden 1, ersetzt diese durch Y, bewegt den LS–Kopf nach links bis zum ersten X und anschließend um eine Position nach rechts zur am weitesten links stehenden 0 und wiederholt die Schleife.

Findet nun M_1 bei der Suche nach einer 1 anstatt dessen ein $\sqcup$, so stoppt M_1 die Berechnung und gibt "nein" als Antwort. Falls M_1, nachdem es eine 1 durch Y ersetzt hat, keine weitere 0 mehr findet, so prüft es, ob noch 1en vorhanden sind, und gibt in diesem Fall die Antwort "nein", andernfalls die Antwort "ja".

Etwas formaler könnten wir M_1 wie folgt angeben:

Zustandsmenge .... $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
Eingabealphabet ... $\Sigma = \{0, 1\}$,
Bandalphabet ..... $\Gamma = \{\sqcup, 0, 1, X, Y\}$,
Endzustände ...... $F = \{q_4\}$,

Überführungsfunktion δ:

Zustand	Symbol				
	$\sqcup$	0	1	X	Y
q_0	–	(q_1, X, R)	–	–	(q_3, Y, R)
q_1	–	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)
q_2	–	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)
q_3	$(q_4, \sqcup, R)$	–	–	–	(q_3, Y, R)
q_4	–	–	–	–	–

Jeden Zustand könnte man sich vorstellen als eine Anweisung oder eine Gruppe von Anweisungen in einem Programm.

- Der Zustand q_0 wird am Start angenommen und auch jeweils unmittelbar vor der Ersetzung der am weitesten links stehenden 0 durch X. Findet M_1 eine 0 vor, so ersetzt es diese durch X, geht über in q_1, und bewegt den LS–Kopf eine Position nach rechts. Findet q_0 ein Y vor, so nimmt M_1 den Zustand q_3 an.

- q_1 wird dazu benutzt, um von links nach rechts nach der ersten 1 zu suchen. Findet M_1 eine 1, so wird sie durch Y ersetzt, und M_1 nimmt den neuen Zustand q_2 an. Wird vor einer 1 ein $\sqcup$ oder ein X gefunden, so bricht die Berechnung ab, und die Eingabe wird zurückgewiesen.

- q_2 wird benutzt um von rechts nach links nach dem ersten X zu suchen. Wird ein X gefunden, so nimmt M_1 den Zustand q_0 an und bewegt den LS–Kopf eine Position nach rechts.

- Im Zustand q_3 wird der LS–Kopf über Y's hinweg nach rechts bewegt um zu prüfen, ob noch 1en auf dem Band vorkommen. Ist das erste Symbol nach den Y's ein $\sqcup$, so nimmt M_1 den Zustand q_4 an und die Eingabefolge wird damit akzeptiert. Andernfalls wird die Eingabefolge zurückgewiesen. ■

Definition 1.4.1: (Fortsetzung) Etwas formaler wird eine *Turing–Maschine* (kurz *TM*) so angegeben wie im obigen Beispiel, nämlich als ein 6–tupel $(Q, \Sigma, \Gamma, q_0, F, \delta)$, wobei Q die Zustandsmenge ist, Σ das Eingabealphabet, Γ das Bandalphabet, q_0 der Startzustand, F die Menge der Endzustände und δ die Überführungsfunktion. δ ist eine partielle Funktion von $Q \times \Gamma$ in $Q \times \Gamma \times \{L, R, S\}$. ■

Definition 1.4.2: Sei M ein TM. Eine *Konfiguration* von M legt den Zustand von M zu einem bestimmten Zeitpunkt fest, also eine Folge von Bandsymbolen $\alpha_1 \ldots \alpha_k$, welche links vom LS–Kopf auf dem Band stehen, eine Folge von Bandsymbolen $\alpha_{k+1} \ldots \alpha_l$, welche rechts vom LS–Kopf auf dem Band stehen einschließlich dem Symbol in der Arbeitszelle (die unendlich vielen $\sqcup$'s am rechten Bandende werden dabei vernachlässigt), sowie den Zustand q der endlichen Kontrolleinheit. Alle anderen Bandsymbole rechts von der Position l sind $\sqcup$. Wir schreiben dafür

$$\alpha_1 \ldots \alpha_k q \alpha_{k+1} \ldots \alpha_l.$$

Also $\alpha_1 \ldots \alpha_k q \alpha_{k+1} \ldots \alpha_l$ und $\alpha_1 \ldots \alpha_k q \alpha_{k+1} \ldots \alpha_l \sqcup$ sind nur zwei verschiedene Notationen für ein und dieselbe Konfiguration.

Eine Konfiguration $\beta_1 \ldots \beta_k q \beta_{k+1} \ldots \beta_m$ geht aus $\alpha_1 \ldots \alpha_l p \alpha_{l+1} \ldots \alpha_m$ hervor,

$$\alpha_1 \ldots \alpha_l p \alpha_{l+1} \ldots \alpha_m \quad \vdash \quad \beta_1 \ldots \beta_k q \beta_{k+1} \ldots \beta_m,$$

wenn $\alpha_i = \beta_i$ für $i \in \{1, \ldots, l, l+2, \ldots, m\}$ und

$$\delta(p, \alpha_{l+1}) = (q, \beta_{k+1}, S) \quad \text{und} \quad k = l,$$

oder

$$\delta(p, \alpha_{l+1}) = (q, \beta_k, R) \quad \text{und} \quad k = l+1,$$

oder

$$\delta(p, \alpha_{l+1}) = (q, \beta_{k+2}, L) \quad \text{und} \quad k = l-1.$$

Eine *Berechnung* der Turing–Maschine M ist eine Folge von Konfigurationen, wobei jeweils die $(i+1)$–te Konfiguration aus der i–ten Konfiguration hervorgeht.

M akzeptiert das Eingabewort $x \in \Sigma^*$, wenn die Berechnung ausgehend von der Konfiguration $q_0 x$ nach endlich vielen Schritten zu einer Konfiguration führt, in welcher der Zustand q ein Element von F ist. ■

Beispiel 1.4.2: Auf der Eingabe 0011 etwa führt die Turing–Maschine M_1 folgende Berechnung aus:

$$\begin{aligned}
&q_0 0011 \vdash X q_1 011 \vdash X 0 q_1 11 \vdash X q_2 0 Y 1 \vdash \\
&q_2 X 0 Y 1 \vdash X q_0 0 Y 1 \vdash X X q_1 Y 1 \vdash X X Y q_1 1 \vdash \\
&X X q_2 Y Y \vdash X q_2 X Y Y \vdash X X q_0 Y Y \vdash X X Y q_3 Y \vdash \\
&X X Y Y q_3 \vdash X X Y Y \sqcup q_4.
\end{aligned}$$

Die Eingabe 0011 wird also von M_1 akzeptiert. ■

Def. 1.4.3: Sei M eine Turing–Maschine . $L(M)$, die *von M akzeptierte Sprache*, ist die Menge aller Eingabewörter x (Folgen von Symbolen aus dem Eingabealphabet Σ), welche von M akzeptiert werden. ■

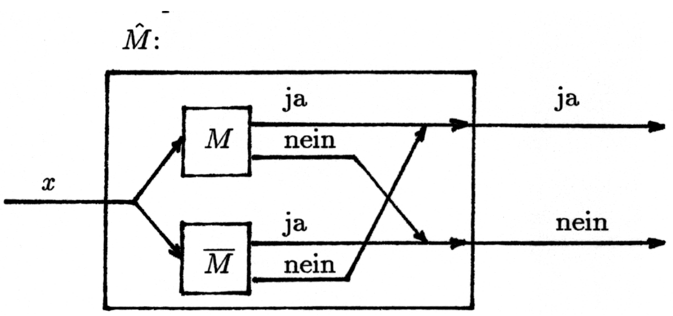
Also $L(M_1) = \{\ 0^n 1^n \ \mid\ n \geq 1\ \}$.

Def. 1.4.4: Eine Sprache L (Menge von Wörtern über einem Alphabet Σ) heißt *rekursiv aufzählbar (r.a.)*, wenn $L = L(M)$ für eine Turing–Maschine M. Eine r.a. Sprache L, welche von einer Turing–Maschine M akzeptiert wird, die auf jedem Eingabewort stoppt, heißt eine *rekursive* Sprache. In diesem Fall sagt man, M *erkennt* L. ■

Satz 1.4.1: *Die rekursiven Sprachen sind genau diejenigen Sprachen L, für die sowohl L als auch das Komplement $\overline{L}$ von L r.a. sind.*

Beweis: Sei L rekursiv. Sei M ein TM, welche immer terminiert und L akzeptiert. Dreht man die Antworten "ja/nein" von M einfach um, so erhält man eine TM $\overline{M}$, welche $\overline{L}$ akzeptiert.

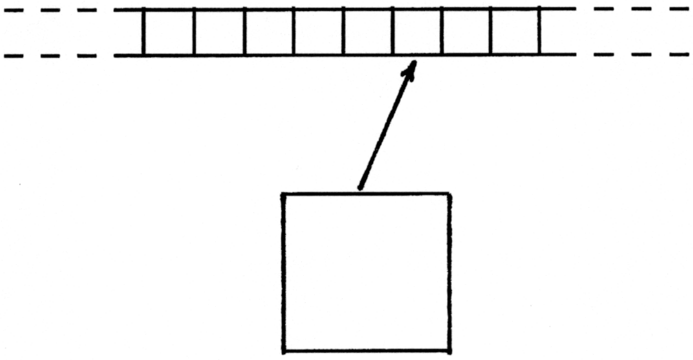
Umgekehrt seien sowohl L als auch $\overline{L}$ r.a. Seien $M, \overline{M}$ TMen, welche L bzw. $\overline{L}$ akzeptieren. Durch Parallelschaltung von M und $\overline{M}$ erhält man eine TM $\hat{M}$, welche immer terminiert und L akzeptiert.



Modifikationen von Turing–Maschinen

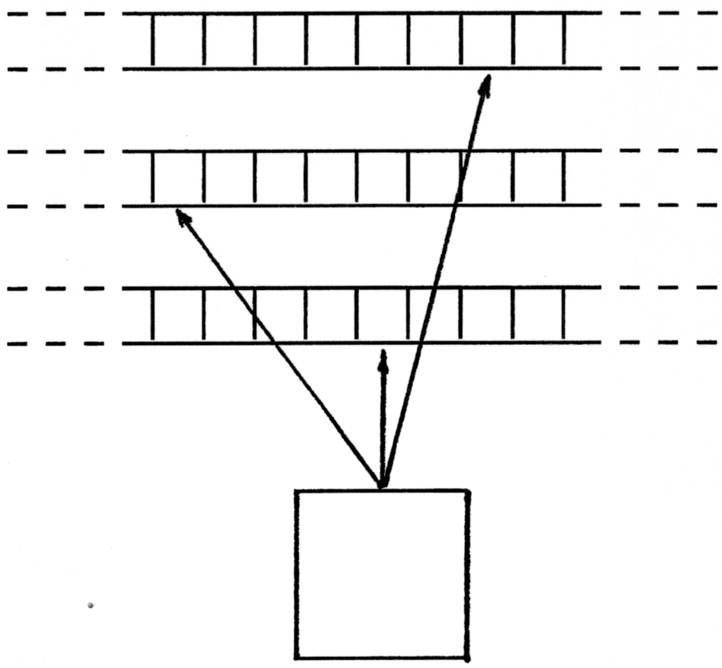
Das Konzept einer Turing–Maschine bietet sich an für eine ganze Reihe von Modifikationen, von denen wir einige kurz beschreiben wollen.

In beide Richtungen unendliches Band:



Das Band ist nicht nur nach rechts unendlich, sondern in beide Richtungen. Ansonsten ändert sich nichts an der Definition der Maschine. Bei Beginn der Berechnung steht das Eingabewort x auf dem Band, das ansonsten nur $\sqcup$'s enthält. Der LS–Kopf zeigt dabei auf das erste Symbol von x.

Turing–Maschine mit mehreren Bändern:



Anstatt eines einzigen Bandes hat die Maschine eine endliche Anzahl von Bändern zur Verfügung. Bei Beginn der Berechnung steht das Eingabewort x auf dem ersten Band, das ansonsten nur $\sqcup$'s enthält. Ebenso enthalten alle übrigen Bänder am Beginn der Berechnung nur $\sqcup$'s.

Abhängig vom Zustand der Kontrolleinheit und von den Symbolen, auf welche die LS–Köpfe zeigen, führt die Maschine M in einem Berechnungsschritt folgende Operationen aus:

– M nimmt einen neuen Zustand an,
– schreibt ein Bandsymbol in die Arbeitszelle eines jeden Bandes, und
– bewegt den LS–Kopf eines jeden Bandes unabhängig um eine Position nach rechts oder nach links, oder läßt ihn stationär.

Nichtdeterministische Turing–Maschine:

Abhängig vom Zustand und vom Symbol in der Arbeitszelle hat die Maschine endlich viele Möglichkeiten einen neuen Zustand anzunehmen, ein Symbol in die Arbeitszelle zu schreiben und den LS-Kopf zu bewegen.

Definition 1.4.5: Eine *nicht–deterministische Turing–Maschine* (kurz *nTM*) M ist definiert wie ein (deterministische) Turing–Maschine , nur liefert δ ausgehend von q

und α eine endliche Menge möglicher Antworten,

$$\delta(q, \alpha) = \{(p_1, \beta_1, X_1), \ldots, (p_m, \beta_m, X_m)\}.$$

Eine Konfiguration geht aus einer anderen hervor, wenn es eine mögliche Antwort der Überführungsfunktion gibt, mittels derer der Übergang hergestellt werden kann. Ausgehend von einer Konfiguration erhalten wir also nicht eine lineare Berechnung, sondern einen Berechnungsbaum.

M akzeptiert die Eingabe x, wenn der Berechnungsbaum ausgehend von der Konfiguration q_0x eine Konfiguration enthält, in welcher der Zustand der Kontrolleinheit q ein Element von F ist. ■

Beispiel 1.4.3: Sei L die Sprache

$$L = \{\, 0^n, (01)^n \mid n \geq 1 \,\}.$$

Die nichtdeterministische Turing–Maschine $M = (Q, \Sigma, \Gamma, q_0, F, \delta)$ akzeptiert L, wobei

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3, q_4\}, \\ \Sigma &= \{0, 1\}, \\ \Gamma &= \{0, 1, \sqcup\}, \\ F &= \{q_4\}, \end{aligned}$$

und δ durch folgende Tabelle gegeben ist:

	$\sqcup$	0	1
q_0	–	$\{(q_1, 0, R), (q_2, 0, R)\}$	–
q_1	$\{(q_4, \sqcup, R)\}$	$\{(q_1, 0, R)\}$	–
q_2	–	–	$\{(q_3, 1, R)\}$
q_3	$\{(q_4, \sqcup, R)\}$	$\{(q_2, 0, R)\}$	–
q_4	–	–	–

■

Satz 1.4.2: *Sei M eine nichtdeterministische Turing–Maschine, welche die Sprache L akzeptiert, also $L = L(M)$. Dann gibt es eine deterministische Turing–Maschine M', welche ebenfalls L akzeptiert, also $L = L(M')$.*

Beweis: Für jedes Paar bestehend aus einem Zustand und einem Bandsymbol von M gibt es eine endliche Anzahl von Wahlmöglichkeiten in der Überführungsfunktion von M. Wir numerieren diese Möglichkeiten durch 1,2,... . Sei r die größte Anzahl von

Wahlmöglichkeiten für ein solches Paar. Dann kann jede endliche Folge von Wahlen dargestellt werden als eine Folge der Nummern 1 bis r. Natürlich entspricht nicht jede Folge solcher Nummern einer Folge von Wahlen, da es für ein gegebenes Paar eventuell weniger als r Wahlmöglichkeiten geben kann.

Die deterministische Turing–Maschine M' hat nun drei Bänder. Das erste Band ist das Eingabeband. Auf dem zweiten Band erzeugt M' systematisch endliche Folgen über $\{1, \ldots, r\}$. Dabei werden zunächst die Folgen der Länge 1 erzeugt, dann diejenigen der Länge 2, usw.

Für jede Folge, welche auf Band 2 erzeugt wird, kopiert M' die Eingabe auf Band 3 und simuliert M auf Band 3, wobei die Folge auf Band 2 dazu benutzt wird, um die Wahlen der entsprechenden Zweige des Berechnungsbaums durchzuführen. Gibt es im Berechnungsbaum von M einen Zweig, der zu einem akzeptierenden Zustand führt, so wird M' schließlich eine Kodierung dieses Zweiges auf Band 2 erzeugen, und bei der Simulierung dieses Zweiges wird M' die Eingabe akzeptieren. Gibt es im Berechnungsbaum von M keinen Zweig, der zu einem akzeptierenden Zustand führt, so wird auch M' nicht akzeptieren.

Also $L(M) = L = L(M')$. ∎

Wir haben also nun einige Möglichkeiten zur Modifikation von Turing–Maschinen betrachtet. Keine dieser Möglichkeiten ist jedoch in der Lage, die Berechnungspotenz einer Turing–Maschine zu steigern. Eine Sprache L, welche von einer auf diese Weise modifizierten Turing–Maschine akzeptiert wird, kann auch von einer Turing–Maschine im ursprünglichen Sinn akzeptiert werden. Für eine genauere Betrachtung dieser Berechnungsäquivalenz verweisen wir auf [HU].

Turing–Maschinen als Generatoren von Sprachen

Definition 1.4.6: Sei M eine Turing–Maschine mit einem oder mehreren Bändern, welche das erste Band als Ausgabeband benützt. M hat ein ausgezeichnetes Bandsymbol #, und M bewegt den LS–Kopf des Ausgabebandes niemals nach links. In diesem Fall ist M geeignet eine Sprache zu erzeugen (generieren). Und zwar ist die *von M erzeugte (generierte) Sprache $G(M)$* die Menge aller Wörter, welche M ohne irgend eine Eingabe zwischen zwei #'s auf das Ausgabeband schreibt. ∎

Satz 1.4.3: *Eine Sprache L über einem Alphabet Σ ist genau dann r.a., wenn L von einer Turing–Maschine generiert werden kann. (Daher auch die Bezeichnung "rekursiv* *aufzählbar".)*

Beweis: Wir wollen einen Beweis dieser Behauptung skizzieren. Angenommen L wird von der Turing–Maschine M generiert, also $G(M) = L$. Wir konstruieren eine Turing–Maschine M' mit einem Band mehr als M. M' simuliert M. Immer wenn M ein Wort x erzeugt, so vergleicht es M' mit dem Eingabewort. Stimmen sie überein, so akzeptiert M' das Eingabewort. Also $L(M') = L$.

Andererseits sei M eine Turing–Maschine, welche L akzeptiert, also $L(M) = L$. Wir konstruieren eine Turing–Maschine M', welche nacheinander alle Paare von positiven ganzen Zahlen (i, j) (in kodierter Form) erzeugt. Sobald das Paar (i, j) erzeugt worden ist, erzeugt M' das i–te Wort x_i über dem Alphabet Σ (in dieser Aufzählung sind die Wörter der Länge nach und Wörter gleicher Länge alphabetisch geordnet) und simuliert j Berechnungsschritte von M auf der Eingabe x_i. Wird x_i von M in nicht mehr als j Schritten akzeptiert, so erzeugt M' das Wort x_i. Somit ist $L = G(M')$. ■

<u>Turing–Maschinen als Berechner von Funktionen</u>

Turing–Maschinen können nicht nur zum Erkennen und Erzeugen von Sprachen eingesetzt werden, sondern sie sind auch in der Lage Funktionen zu berechnen.

Definition 1.4.7: Seien Σ_1 und Σ_2 zwei Alphabete und f eine (partielle) Funktion von $(\Sigma_1^*)^k$ in Σ_2^* (Σ_1^* ist die Menge aller Wörter über Σ_1). Eine Turing–Maschine M *berechnet* die Funktion f falls

– $\Sigma_1 \cup \Sigma_2$ eine Teilmenge des Bandalphabets von M ist, und

– für jedes $x = (x_1, \ldots, x_k) \in (\Sigma_1^*)^k$, wenn $f(x_1, \ldots, x_k) = y$ dann stoppt M auf dem Eingabewort $x_1 \sqcup \ldots \sqcup x_k$ mit Ausgabe (Bandinhalt nach Ende der Berechnung) y und wenn $f(x_1, \ldots, x_k)$ undefiniert ist, dann stoppt M auf dem Eingabewort $x_1 \sqcup \ldots \sqcup x_k$ nicht.

Eine (partielle) Funktion, welche von einer Turing–Maschine berechnet werden kann, heißt *(partiell) Turing–berechenbar*, abgekürzt *(p.)T.b.* ■

Beispiel 1.4.4: Ein Beispiel einer T.b. Funktion ist die modifizierte Differenz $\dot{-}$, welche für zwei natürliche Zahlen m, n den Wert $m - n$ liefert falls $m \geq n$ und den Wert 0 sonst. $\dot{-}$ wird von der Turing–Maschine M_2 berechnet mit

Zustandsmenge .... $Q = \{q_0, \ldots, q_6\}$,
Eingabealphabet ... $\Sigma = \{0\}$,

Bandalphabet ..... $\Gamma = \{\sqcup, 0, 1\}$,
Endzustände ...... $F = \{\}$.

Die Eingabeparameter m, n werden kodiert als

$$\underbrace{0\ldots0}_{m \text{ mal}}\sqcup\underbrace{0\ldots0}_{n \text{ mal}}.$$

Die Überführungsfunktion δ von M_2 arbeitet wie folgt:

(1) Start. Ersetze die führende 0 durch $\sqcup$. Ist keine 0 vorhanden, so lösche den Rest des Bandes.
$\delta(q_0, 0) = (q_1, \sqcup, R)$, $\delta(q_0, \sqcup) = (q_5, \sqcup, R)$.

(2) Bewege den LS–Kopf nach rechts und suche nach dem ersten $\sqcup$. Überschreibe dieses durch 1.
$\delta(q_1, 0) = (q_1, 0, R)$, $\delta(q_1, \sqcup) = (q_2, 1, R)$, $\delta(q_1, 1) = (q_2, 1, R)$.

(3) Bewege den LS–Kopf nach rechts über 1en hinweg bis zur ersten 0. Überschreibe diese durch 1.
$\delta(q_2, 1) = (q_2, 1, R)$, $\delta(q_2, 0) = (q_3, 1, L)$.

(4) Gehe nach links bis zum ersten $\sqcup$. Nimm Zustand q_0 an und wiederhole den Zyklus.
$\delta(q_3, 0) = (q_3, 0, L)$, $\delta(q_3, 1) = (q_3, 1, L)$, $\delta(q_3, \sqcup) = (q_0, \sqcup, R)$.

(5) Wird im Zustand q_2 ein $\sqcup$ vor einer 0 gefunden, so nimm Zustand q_4 an und gehe nach links, wobei alle 1en durch $\sqcup$'s überschrieben werden, bis zum ersten $\sqcup$. Ändere dieses $\sqcup$ wieder zu einer 0 ab, nimm Zustand q_6 an und stoppe.
$\delta(q_2, \sqcup) = (q_4, \sqcup, L)$, $\delta(q_4, 1) = (q_4, \sqcup, L)$, $\delta(q_4, 0) = (q_4, 0, L)$, $\delta(q_4, \sqcup) = (q_6, 0, R)$.

(6) Wird im Zustand q_0 eine 1 anstatt einer 0 gefunden, so ist offensichtlich der erste Block von 0en abgearbeitet. Nimm Zustand q_5 an und lösche den Rest des Bandes. Anschließend nimm Zustand q_6 an und stoppe.
$\delta(q_0, 1) = (q_5, \sqcup, R)$, $\delta(q_5, 0) = (q_5, \sqcup, R)$, $\delta(q_5, 1) = (q_5, \sqcup, R)$, $\delta(q_5, \sqcup) = (q_6, \sqcup, R)$.

Zustand	Symbol		
	0	1	$\sqcup$
q_0	$(q_1, \sqcup, R)$	$(q_5, \sqcup, R)$	$(q_5, \sqcup, R)$
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	$(q_2, 1, R)$
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	$(q_4, \sqcup, L)$

Zustand	Symbol 0	1	$\sqcup$
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	$(q_1, \sqcup, R)$
q_4	$(q_4, 0, L)$	$(q_4, \sqcup, L)$	$(q_6, 0, R)$
q_5	$(q_5, \sqcup, R)$	$(q_5, \sqcup, R)$	$(q_6, 0, R)$
q_6	$-$	$-$	$-$

Berechnung von M_2 auf $00 \sqcup 0 \quad (2 \dot{-} 1)$:

$$\begin{aligned}
&q_0 00 \sqcup 0 \vdash \sqcup q_1 0 \sqcup 0 \vdash \sqcup 0 q_1 \sqcup 0 \vdash \sqcup 01 q_2 0 \vdash \\
&\sqcup 0 q_3 11 \vdash \sqcup q_3 011 \vdash q_3 \sqcup 011 \vdash \sqcup q_0 011 \vdash \\
&\sqcup \sqcup q_1 11 \vdash \sqcup \sqcup 1 q_2 1 \vdash \sqcup \sqcup 11 q_2 \vdash \sqcup \sqcup 1 q_4 1 \vdash \\
&\sqcup \sqcup q_4 1 \vdash \sqcup q_4 \vdash \sqcup 0 q_6.
\end{aligned}$$

Berechnung von M_2 auf $0 \sqcup 00 \quad (1 \dot{-} 2)$:

$$\begin{aligned}
&q_0 0 \sqcup 00 \vdash \sqcup q_1 \sqcup 00 \vdash \sqcup 1 q_2 00 \vdash \sqcup q_3 110 \vdash \\
&q_3 \sqcup 110 \vdash \sqcup q_0 110 \vdash \sqcup \sqcup q_5 10 \vdash \sqcup \sqcup \sqcup q_5 0 \vdash \\
&\sqcup \sqcup \sqcup \sqcup q_5 \vdash \sqcup \sqcup \sqcup \sqcup \sqcup q_6.
\end{aligned}$$

■

Alle üblichen arithmetischen Funktionen wie $m + n$, $m \cdot n$, m^n, $n!$, $\lceil \log_2 n \rceil$, etc. sowie Kombinationen davon sind T.b.

Zwischen r.a. Sprachen und p.T.b. Funktionen besteht ein enger Zusammenhang.

Satz 1.4.4: *Ist f eine partielle Funktion von $(\Sigma_1^*)^k$ nach Σ_2^*, so ist f genau dann p.T.b., wenn $L_f = \{ \ (x_1, \ldots, x_k, y) \ \mid \ f(x_1, \ldots, x_k) = y \ \}$ r.a. ist.*

Beweis: Übung. ■

Im folgenden befassen wir uns mit der Äquivalenz zwischen dem RAM/RASP Modell und dem Turing–Maschinen Modell.

Satz 1.4.5: *Jede Turing–Maschine (mit mehreren Bändern) kann durch eine RAM simuliert werden. Also jede Sprache (Funktion), die von einer Turing–Maschine akzeptiert (berechnet) werden kann, kann auch von einer RAM akzeptiert (berechnet) werden.*

Beweis: (Skizze) Sei M eine k–Band Turing–Maschine . In der M simulierenden RAM R ist jede Bandzelle von M in einem Register abgespeichert, und zwar ist die i–te Zelle des j–ten Bandes im Register $ki + j + c$ abgespeichert. Dabei ist c so angelegt, daß R noch Arbeitsspeicher zur Verfügung hat, u.a. zur Abspeicherung der Positionen der k LS–Köpfe von M. Bandzellen von M können von R gelesen werden durch indirekte Adressierung mittels der Register, welche die Positionen der LS–Köpfe beinhalten. ■

Satz 1.4.6: *Jede RAM kann durch eine Turing–Maschine simuliert werden. Also jede Sprache (Funktion), die von einer RAM akzeptiert (berechnet) werden kann, kann auch von einer Turing–Maschine akzeptiert (berechnet) werden.*

Beweis: (Skizze) Die Turing–Maschine M, die die RAM R simuliert, besitzt neben den Ein– und Ausgabebändern noch drei zusätzliche Bänder B_3, B_4, B_5. B_3 dient zur Darstellung des Speichers von R und hat die Form

#	#	#	i_1	#	c_1	#	#	i_2#	c_2	#	#	$\cdots$	i_k	#	c_k	#	#	$\sqcup$	$\ldots$

Dabei sind $i_1, \ldots, i_k$ die Indizes derjenigen Register von R, welche nicht 0 enthalten und $c_j = c(i_j)$; i_j, c_j sind in Binärdarstellung auf B_3 wiedergegeben, für $1 \le i \le k$. Der Inhalt des Akkumulators von R ist in Binärdarstellung auf B_4 dargestellt, und B_5 wird als Arbeitsband verwendet.

Jeder Rechenschritt von R entspricht einer endlichen Folge von Zuständen der Turing–Maschine M. Als Beispiel betrachten wir die RAM Instruktionen `ADD *i` und `STORE i`. Die Instruktion `ADD *i` wird von M folgendermaßen simuliert:

1. Suche auf B_3 nach einer Eintragung für das RAM Register i, d.h. eine Folge $\#\#i'\#$, wobei i' die Binärdarstellung von i ist. Ist eine solche Eintragung vorhanden, so schreibe die nächste Zahl, welche der Inhalt von Register i ist, auf B_5; andernfalls halte (der Inhalt von Register i ist 0, die indirekte Adressierung kann also nicht durchgeführt werden).
2. Suche auf B_3 nach einer Eintragung für das RAM Register, dessen Index auf B_5 steht. Ist eine solche Eintragung vorhanden, so schreibe den Inhalt dieses Registers auf B_5; andernfalls schreibe 0 auf B_5.
3. Addiere die Zahl auf B_5 zum Inhalt des Akkumulators, der auf B_4 steht.

Die Instruktion `STORE i` wird von M folgendermaßen simuliert:

1. Suche auf B_3 nach einer Eintragung für das RAM Register i, also $\#\#i'\#$.

2. Ist eine solche Eintragung vorhanden, so kopiere den Bandinhalt von B_3 rechts von $\#\#i'\#$, außer der unmittelbar nächsten Zahl ($c(i)$), auf B_5. Kopiere den Inhalt von B_4 (Akkumulator) auf B_3 unmittelbar rechts von $\#\#i'\#$ und unmittelbar daran den Inhalt von B_5.
3. Ist eine Eintragung für Register i nicht vorhanden, so schreibe $i'\#$ auf B_3 unmittelbar nach dem letzten von $\sqcup$ verschiedenen Symbol, anschließend den Inhalt von B_4 (Akkumulator) und schließlich $\#\#$.

Auf ganz ähnliche Weise können auch die übrigen RAM Instruktionen simuliert werden.

■

Die Menge der (partiell) T.b. (zahlentheoretischen) Funktionen ist nicht nur identisch mit der Menge der (partiell) RAM berechenbaren Funktionen, sie ist auch identisch mit der Menge der (partiell) rekursiven Funktionen, oder mit der Menge der WHILE-berechenbaren Funktionen (siehe etwa [DW], [Sch]). Alle diese mathematischen Modelle liefern dieselbe Klasse algorithmischer Funktionen. Diese Klasse scheint also in gewisser Weise eine natürliche zu sein. Andererseits sind alle (bisher betrachteten) Funktionen, die man intuitiv als berechenbar bezeichnen würde, tatsächlich rekursiv. Die meisten Mathematiker und Computerwissenschafter sind daher der Ansicht, daß mit dem formalen Begriff der rekursiven bzw. Turing–berechenbaren Funktion der intuitive Begriff der "algorithmischen" Funktion charakterisiert ist. Diese Überzeugung wurde zum ersten Mal von A. Church ausgesprochen und heißt nach ihm die *Church'sche These*:

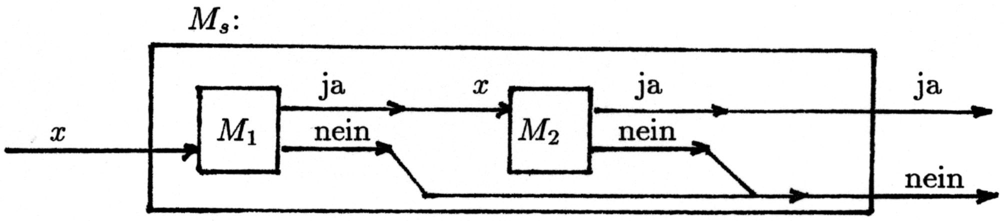
> *Eine Funktion ist berechenbar genau dann wenn sie Turing–berechenbar (rekursiv) ist.*

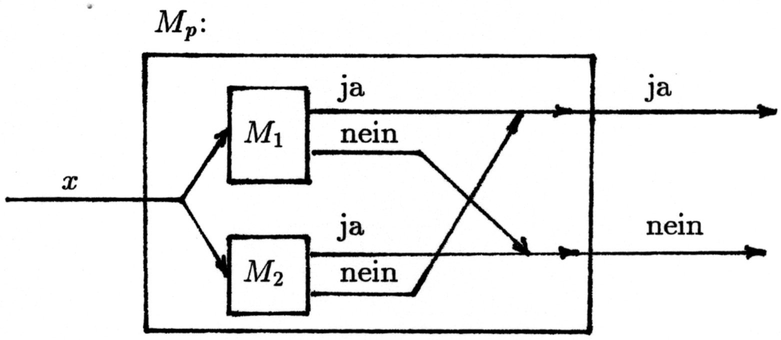
Im strengen Sinne beweisen läßt sich die Church'sche These natürlich nicht, da man auf einem intuitiven Begriff keinen Beweis aufbauen kann. Aufgrund der Church'schen These versteht man heutzutage unter einem Algorithmus eine Turing–Maschine . Wenn wir also sagen, daß sich ein Problem algorithmisch lösen läßt, so meinen wir streng genommen damit, daß es eine Turing–Maschine gibt, die das Problem löst.

Definition 1.4.8: Ein *Algorithmus* ist eine Turing–Maschine , die auf jeder Eingabe terminiert. ■

Bisher haben wir nur Algorithmen definiert, welche ganze Zahlen als Eingaben nehmen und wiederum ganze Zahlen produzieren. Im intuitiven Sinne kennt man natürlich auch Algorithmen mit ganz anderen Ein– und Ausgabebereichen. Zur Formalisierung solcher Algorithmen gibt es im Prinzip zwei Ansätze. Beim ersten Ansatz ordnet man jedem Element der Ein– und Ausgabemenge eine natürliche Zahl zu und kann dann das herkömmliche Konzept des zahlentheoretischen Algorithmus verwenden. Beim zweiten Ansatz verlangt man nur, das sich die Elemente der Ein– und Ausgabemenge als Wörter über einem Alphabet schreiben lassen. Solche Wörter können nun von einer Turing–Maschine gelesen bzw. geschrieben werden und man erhält auf diese Weise eine Verallgemeinerung des Begriffs "Algorithmus".

Übung: Seien M_1, M_2 Turing–Maschinen. Dann gibt es Turing–Maschinen M_s, M_p mit dem Verhalten





Übung: Das Komplement einer rekursiven Sprache ist rekursiv.

Übung: Die Vereinigung zweier rekursiver Sprachen ist rekursiv. Die Vereinigung zweier r.a. Sprachen ist r.a.

1.5. Rekursive Funktionen

Im vorangegangenen Abschnitt haben wir gesehen, wie Turingmaschinen zur Berechnung von Funktionen, insbesondere zahlentheoretischer Funktionen (von $\mathbb{N}^k$ nach $\mathbb{N}$) eingesetzt werden können. Wir haben auf diese Weise den Begriff der Turing-berechenbaren Funktion erhalten. Die Äquivalenz von Turingmaschinen und RAMs kann natürlich von der Akzeptierung von Sprachen auf die Berechnung von Funktion ausgedehnt werden, sodaß jede Turing-berechenbare Funktion auch von einer RAM berechnet werden kann. Wir sprechen also in Hinkunft nur mehr von berechenbaren Funktionen.

In diesem Abschnitt wollen wir einen anderen, rein mathematisch-logischen Zugang zum Begriff der berechenbaren Funktion behandeln, nämlich den Begriff der rekursiven Funktion. Wiederum kann man feststellen, daß der Begriff der berechenbaren Funktion mit demjenigen der rekursiven Funktion übereinstimmt. Somit haben wir eine weitere Bestätigung der Church'schen These vorliegen. Wir werden hier weitgehend auf Beweise der Äquivalenz von Berechenbarkeit und Rekursivität verzichten. Der Leser ist aufgefordert, sich mithilfe der Literaturliste jeweils Rechenschaft über diese Äquivalenz zu geben.

Immer, wenn wir in diesem Abschnitt von einer Funktion sprechen, so meinen wir eine zahlentheoretische Funktion im obigen Sinne.

Definition 1.5.1: Sei f eine Funktion von k Variablen, und seien $g_1, \ldots, g_k$ Funktionen von n Variablen. Sei

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)).$$

Dann entsteht h aus f und $g_1, \ldots, g_k$ durch *Komposition*. ■

Natürlich müssen die Funktionen $f, g_1, \ldots, g_k$ nicht total sein, sondern können auch partiell sein. $h(x_1, \ldots, x_n)$ ist definiert, wenn alle $z_i = g_i(x_1, \ldots, x_n)$ definiert sind, und auch $f(z_1, \ldots, z_k)$ definiert ist.

Definition 1.5.2: Sei f eine Funktion von n Variablen und g eine Funktion von $n+2$ Variablen. Sei

$$h(m, x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n), & \text{falls } m = 0, \\ g(m-1, h(m-1, x_1, \ldots, x_n), x_1, \ldots, x_n), & \text{falls } m > 0. \end{cases}$$

Dann entsteht h aus f und g durch *Rekursion*. ■

Um die Klasse der rekursiven Funktionen einzuführen, werden wir Funktionen mittels Komposition und Rekursion (und der noch einzuführenden Minimalisierung) kombinieren. Wir brauchen aber auch eine Basis von Grundfunktionen, von denen wir ausgehen können.

Definition 1.5.3: Wir betrachten folgende Funktionen:

Nullfunktion:	$\mathrm{null}() = 0$,
Nachfolgerfunktion (successor):	$\mathrm{succ}(x) = x + 1$,
Projektionsfunktionen:	$\mathrm{proj}_i^n(x_1, \ldots, x_n) = x_i,\ 1 \le i \le n$.

Die Funktionen $\mathrm{null}, \mathrm{succ}, \mathrm{proj}_i^n$ heißen *Grundfunktionen*. ■

Damit haben wir nun die Voraussetzungen geschaffen, um zunächst den Begriff der primitiv rekursiven Funktion einzuführen.

Definition 1.5.4: Die Klasse $\mathcal{PR}$ der *primitiv rekursiven Funktionen* ist induktiv wie folgt definiert:

(i) die Grundfunktionen gehören zu $\mathcal{PR}$,

(ii) alle Funktionen, welche aus Funktionen in $\mathcal{PR}$ durch Komposition und Rekursion erzeugt werden, gehören zu $\mathcal{PR}$.

Keine andere Funktion gehört zu $\mathcal{PR}$. ■

Beispiel 1.5.1: Viele der bekannten zahlentheoretischen Funktionen sind primitiv rekursiv. Wir weisen das für einige Funktionen nach.

(a) "$x + y$" ist primitiv rekursiv: die Additionsfunktion $f(x, y) = x + y$ ist rekursiv definiert als

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

Diese Definition kann geschrieben werden als

$$f(x, y) = \begin{cases} \mathrm{proj}_1^1(y), & \text{falls } x = 0, \\ \mathrm{succ}(\mathrm{proj}_2^3(x - 1, f(x - 1, y), y)), & \text{falls } x > 0. \end{cases}$$

"$x+y$" entsteht also aus den Grundfunktionen durch Rekursion und Komposition.

(b) "$x \cdot y$" ist primitiv rekursiv: die Multiplikationsfunktion $g(x, y) = x \cdot y$ ist rekursiv definiert als

$$g(0, y) = 0,$$
$$g(x + 1, y) = g(x, y) + y.$$

Diese Definition kann geschrieben werden als

$$g(x,y) = \begin{cases} \mathrm{null}(y), & \text{falls } x = 0, \\ r(x-1, g(x-1,y), y), & \text{falls } x > 0. \end{cases}$$

Dabei sei $f(x,y) = x + y$ wie oben, und

$$r(y,v,w) = f(\mathrm{proj}_2^3(u,v,w), \mathrm{proj}_3^3(u,v,w)).$$

Die unäre Nullfunktion $\mathrm{null}(y) = 0$ entsteht aus null() durch Komposition (dabei ist $k = 0$ in Def. 1.5.1). "$x \cdot y$" entsteht also aus den Grundfunktionen durch Rekursion und Komposition.

(c) "$x!$" ist primitiv rekursiv: die Rekursionsgleichungen sind

$$0! = 1,$$
$$(x+1)! = x! \cdot \mathrm{succ}(x).$$

Die Funktion $h(x) = x!$ kann also geschrieben werden als

$$h(x) = \begin{cases} \mathrm{succ}(\mathrm{null}()), & \text{falls } x = 0, \\ s(x-1, h(x-1)), & \text{falls } x > 0, \end{cases}$$

wobei

$$s(x_1, x_2) = \mathrm{succ}(x_1) \cdot x_2.$$

Die Faktoriellenfunktion "$x!$" ist also primitiv rekursiv.

In den folgenden Beispielen primitiv rekursiver Funktionen überlassen wir es dem Leser zu überprüfen, daß die Rekursionsgleichungen in die exakte Gestalt gebracht werden können, wie sie von der Definition gefordert wird.

(d) "x^y": die Rekursionsgleichungen sind

$$x^0 = 1,$$
$$x^{y+1} = x^y \cdot x.$$

(e) "pred(x)": die Vorgängerfunktion pred(x) ist folgendermaßen definiert:

$$\mathrm{pred}(x) = \begin{cases} 0, & \text{falls } x = 0, \\ x - 1, & \text{sonst.} \end{cases}$$

(f) Die modifizierte Differenz "$x \dot{-} y$" ist definiert als

$$x \dot{-} y = \begin{cases} x - y, & \text{falls } x \geq y, \\ 0, & \text{falls } x < y. \end{cases}$$

Die Rekursionsgleichungen sind

$$\begin{aligned} x \dot{-} 0 &= x, \\ x \dot{-} (t+1) &= \mathrm{pred}(x \dot{-} t). \end{aligned}$$

(g) Die Abstandsfunktion $| x - y |$ ist definiert als Absolutwert der Differenz zwischen x und y. Sie kann dargestellt werden als

$$| x - y | = (x \dot{-} y) + (y \dot{-} x). \qquad \blacksquare$$

Unter einem *Prädikat* verstehen wir einfach eine Funktion, welche nur die Werte 0 oder 1 annimmt. Interpretieren wir 1 als "wahr" und 0 als "falsch", so erhalten wir die übliche Vorstellung von einem Prädikat als boolescher Funktion. Da also Prädikate nichts anderes als spezielle zahlentheoretische Funktionen sind, können wir problemlos von primitiv rekursiven Prädikaten sprechen.

Beispiel 1.5.2: Die folgenden Prädikate sind primitiv rekursiv.

(a) Das Prädikat istnull(x) ist definiert als

$$\mathrm{istnull}(x) = \begin{cases} 1, & \text{falls } x = 0, \\ 0, & \text{falls } x \neq 0. \end{cases}$$

Wir können auch schreiben istnull(x) $= 1 \dot{-} x$.

(b) Das Gleichheitsprädikat "$x = y$" entspricht der Funktion

$$d(x, y) = \begin{cases} 1, & \text{falls } x = y, \\ 0, & \text{falls } x \neq y. \end{cases}$$

Das Gleichheitsprädikat ist also primitiv rekursiv, da wir schreiben können

$$d(x, y) = \mathrm{istnull}(| x - y |).$$

(c) Das Prädikat "$x \leq y$" ist einfach die Funktion istnull($x \dot{-} y$). $\blacksquare$

Satz 1.5.1: *Sind P, Q primitiv rekursive Prädikate, dann sind auch $\neg P, P \vee Q, P \wedge Q$ primitiv rekursiv.*

Beweis: $\neg P = \mathrm{istnull}(P)$, somit ist $\neg P$ primitiv rekursiv.

Die Konjunktion ist primitiv rekursiv wegen $P \wedge Q = P \cdot Q$.

Schließlich sieht man aus der Regel von De Morgan,

$$P \vee Q \iff \neg(\neg P \wedge \neg Q),$$

daß auch die Disjunktion primitiv rekursiv ist. ■

Satz 1.5.2: *Seien g und h primitiv rekursive Funktionen und P ein primitiv rekursives Prädikat. Sei f die folgendermaßen definierte Funktion*

$$f(x_1, \ldots, x_n) = \begin{cases} g(x_1, \ldots, x_n) & \textit{falls } P(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n) & \textit{sonst.} \end{cases}$$

Dann ist auch f primitiv rekursiv.

Beweis: Das sieht man sofort aus der Beziehung

$$\begin{aligned} &f(x_1, \ldots, x_n) = \\ &g(x_1, \ldots, x_n) \cdot P(x_1, \ldots, x_n) + h(x_1, \ldots, x_n) \cdot (\neg P(x_1, \ldots, x_n)). \end{aligned}$$

■

Satz 1.5.3: *Sei $f(t, x_1, \ldots, x_n)$ eine primitiv rekursive Funktion. Dann sind auch die folgendermaßen definierten Funktionen g und h primitiv rekursiv.*

$$\begin{aligned} g(y, x_1, \ldots, x_n) &= \sum_{t=0}^{y} f(t, x_1, \ldots, x_n), \\ h(y, x_1, \ldots, x_n) &= \prod_{t=0}^{y} f(t, x_1, \ldots, x_n). \end{aligned}$$

Beweis: Es gelten die Rekursionsgleichungen

$$\begin{aligned} g(0, x_1, \ldots, x_n) &= f(0, x_1, \ldots, x_n), \\ g(t+1, x_1, \ldots, x_n) &= g(t, x_1, \ldots, x_n) + f(t+1, x_1, \ldots, x_n). \end{aligned}$$

Da die Addition primitiv rekursiv ist, ist auch g primitiv rekursiv.

Analog verfährt man mit

$$\begin{aligned} h(0, x_1, \ldots, x_n) &= f(0, x_1, \ldots, x_n), \\ h(t+1, x_1, \ldots, x_n) &= h(t, x_1, \ldots, x_n) \cdot f(t+1, x_1, \ldots, x_n). \end{aligned}$$

(Warum ist es nicht zielführend, den Beweis durch Induktion über y zu versuchen?) ■

Satz 1.5.4: *Sei $P(t, x_1, \ldots, x_n)$ ein primitiv rekursives Prädikat. Dann sind auch die folgenden Prädikate primitiv rekursiv,*

$$(\forall t)_{\leq y} P(t, x_1, \ldots, x_n) \quad \text{und} \quad (\exists t)_{\leq y} P(t, x_1, \ldots, x_n).$$

Beweis: Das folgt sofort aus der Beobachtung

$$(\forall t)_{\leq y} P(t, x_1, \ldots, x_n) \iff \Big[\prod_{t=0}^{y} P(t, x_1, \ldots, x_n)\Big] = 1$$

und

$$(\exists t)_{\leq y} P(t, x_1, \ldots, x_n) \iff \Big[\sum_{t=0}^{y} P(t, x_1, \ldots, x_n)\Big] \neq 0. \qquad \blacksquare$$

Beispiel 1.5.3: Die folgenden Prädikate sind primitiv rekursiv.

(a) "$y \mid x$", also das Prädikat "y teilt x".

$$y \mid x \iff (\exists t)_{\leq x}(y \cdot t = x).$$

(b) "Prime(x)", also das Prädikat "x ist eine Primzahl".

$$\mathrm{Prime}(x) \iff (x \geq 2) \wedge (\forall t)_{\leq x}\{t = 1 \vee t = x \vee \neg(t \mid x)\}. \qquad \blacksquare$$

Wie wir also gesehen haben, umfaßt die Klasse der primitiv rekursiven Funktionen eine große Anzahl der typischen zahlentheoretischen Funktionen und Prädikate. Alle diese primitiv rekursiven Funktionen sind (Turing-)berechenbar. Es gibt aber andererseits berechenbare Funktionen, welche nicht primitiv rekursiv sind. Ein klassisches Beispiel dafür ist die Ackermannfunktion

$$\begin{aligned} \mathrm{ack}(0, y) &= y + 1, \\ \mathrm{ack}(x, 0) &= \mathrm{ack}(x - 1, 1), \\ \mathrm{ack}(x, y) &= \mathrm{ack}(x - 1, \mathrm{ack}(x, y - 1)), \quad \text{sonst}. \end{aligned}$$

Einen Beweis dieser Tatsache findet man etwa in [DW]. Um auch solche Funktionen behandeln zu können, muß man die Klasse $\mathcal{PR}$ erweitern um das Konstruktionsprinzip der Minimalisierung. Dadurch erhält man dann die Klasse der rekursiven Funktionen, welche dann tatsächlich mit den berechenbaren Funktionen übereinstimmt.

Definition 1.5.5: Sei $P(y, x_1, \ldots, x_n)$ ein Prädikat und sei $\min_y P(y, x_1, \ldots, x_n)$ die (partielle) Funktion, welche den kleinsten Wert y ergibt, sodaß das Prädikat $P(y, x_1, \ldots, x_n)$ wahr ist, falls ein solches y existiert. Andernfalls ist $\min_y P(y, x_1, \ldots, x_n)$ undefiniert. $\blacksquare$

Definition 1.5.6: Die Klasse $\mathcal{R}$ der *rekursiven Funktionen* ist induktiv wie folgt definiert:

(i) die Grundfunktionen gehören zu $\mathcal{R}$,

(ii) alle Funktionen, welche aus Funktionen (und Prädikaten) in $\mathcal{R}$ durch Komposition, Rekursion und Minimalisierung erzeugt werden, gehören zu $\mathcal{R}$.

Keine andere Funktion gehört zu $\mathcal{R}$. ∎