Eingabe: Eine $k$-NDTM $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit einer platz-konstruierbaren Speicherplatzkomplexität $S(n)$ und ein Wort $w \in I^*$ mit $|w| = n$

Verfahren: Aufruf der Funktion `space_Simulation (`*TM*`, `*w*`)`

Ausgabe: `TRUE`, falls $w \in L(TM)$, `FALSE` sonst.

Die Funktion `space_Simulation` wird in Pseudocode beschrieben; sie besitzt zwei Formalparameter `TM` und `w`, über die die Beschreibung einer $k$-NDTM und ein Eingabewort für diese Turingmaschine eingegeben werden. Innerhalb von `space_Simulation` wird eine Funktion `test` verwendet, die als Parameter zwei Konfigurationen `K1` und `K2` und eine natürliche Zahl `i` hat. Auf die genaue syntaktische Spezifikation soll hier verzichtet werden.

```
FUNCTION space_Simulation (TM : ...;
                           w  : ...) : BOOLEAN;

VAR Kf : ...;
    K0 : ...;
    OK : BOOLEAN;

     FUNCTION test (K1 :...;
                    K2 : ...;
                    i  : INTEGER) : BOOLEAN;
       VAR resultat : BOOLEAN;
           K3       : ...;
       BEGIN { test }
         resultat := FALSE;
         IF i = 1 THEN BEGIN
                         IF (K1 ⇒ K2) OR (K1 = K2)
                         THEN resultat := TRUE;
                       END
                  ELSE BEGIN
                         Für jede Konfiguration K3 = (q,(a_1,i_1),...,(a_k,i_k))
                         mit |a_j| ≤ S(n) und 1 ≤ i_j ≤ S(n) DO
                         BEGIN
                           resultat := test (K1, K3, (i+1) DIV 2)
                                       AND
                                       test (K3, K2, i DIV 2);
                           IF resultat = TRUE THEN Break;
                         END;
                       END;
         test := resultat;
      END    { test };
```

```
BEGIN { space_Simulation }
  K0 := (q0,(w,1),(e,1),...,(e,1))
  OK := FALSE;
  FOR_EACH Endkonfiguration Kf = (qf,(a1,|a1|+1),...,(ak,|ak|+1)) mit |aj| ≤ S(n)
    DO BEGIN
         OK := test (K0, Kf, c^S(n));
         IF OK THEN Break;
       END;
 space_Simulation := OK;
END   { space_Simulation };
```

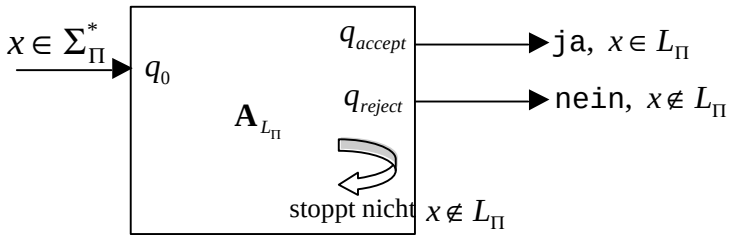
Eine genauere Untersuchung der Aufrufe der rekursiven Funktion `test` zeigt, daß bei jedem Aufruf innerhalb von `test` der dritte Parameter im wesentlichen halbiert wird. Der maximale Wert des dritten Parameters ist `i` = $c^{S(n)}$. Daher werden zu keinem Zeitpunkt der (rekursiven) Abarbeitung von `test` mehr als $1+\log(\lceil c^{S(n)} \rceil) \in O(S(n))$ viele Aktivierungsrecords auf dem Stack benötigt. Jeder Aktivierungsrecord hat eine Größe der Ordnung $O(S(n))$, so daß der Speicherplatzbedarf von der Ordnung $O(S^2(n))$ ist. ///

Der Begriff des Nichtdeterminismus läßt sich auf Algorithmen (formuliert als Programm in einer Programmiersprache, vgl. Kapitel 2.3) übertragen und führt auf die Definition **eines nichtdeterministischen Algorithmus unter Einsatz eines Verifizierers**. In Kapitel 1.3 und Kapitel 2.1 wird ein Algorithmus für ein Entscheidungsproblem wie folgt definiert (da der Algorithmus als Programm in einer Programmiersprache beschrieben wird, handelt es sich um einen *deterministischen* Algorithmus).

Ein **deterministischer Algorithmus $\mathbf{A}_{L_\Pi}$ für ein Entscheidungsproblem** $\Pi$ mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$

Ausgabe: `ja` (accept), falls $x \in L_\Pi$ gilt
`nein` (reject), falls $x \notin L_\Pi$ gilt
falls $\mathbf{A}_{L_\Pi}$ bei einer Eingabe $x \in \Sigma_\Pi^*$ nicht hält, gilt ebenfalls $x \notin L_\Pi$.

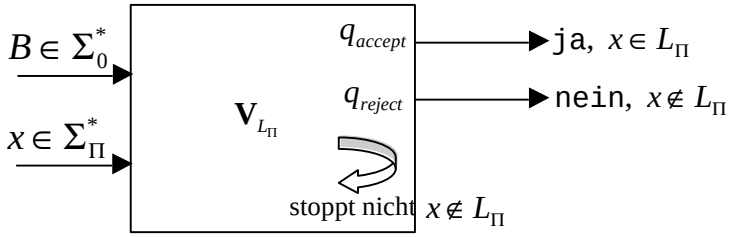



Ein **Verifizierer für das Entscheidungsproblem** $\Pi$ über einem Alphabet $\Sigma_\Pi$ ist ein deterministischer Algorithmus $\mathbf{V}_{L_\Pi}$, der eine Eingabe $x \in \Sigma_\Pi^*$ und eine **Zusatzinformation** (einen **Beweis**) $B \in \Sigma_0^*$ lesen kann und die Frage „$x \in L_\Pi$?“ mit Hilfe des Beweises $B$ entscheidet. Die Bereitstellung des Beweises $B$ ist nicht Aufgabe des Verifizierers; $B$ wird „von außen“ vorgegeben. Eventuell stoppt der Verifizierer bei Eingabe von $x \in \Sigma_\Pi^*$ nicht. **Der Verifizierer repräsentiert also die Verifikationsphase** im Nichtstandardmodell einer nichtdeterministischen Turingmaschine; der erforderliche Beweis wird zuvor vom Ratemodul auf nichtdeterministische Weise erzeugt.

Ein **Verifizierer** $\mathbf{V}_{L_\Pi}$ **für das Entscheidungsproblem** $\Pi$ mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ hat die Schnittstellen und die Form

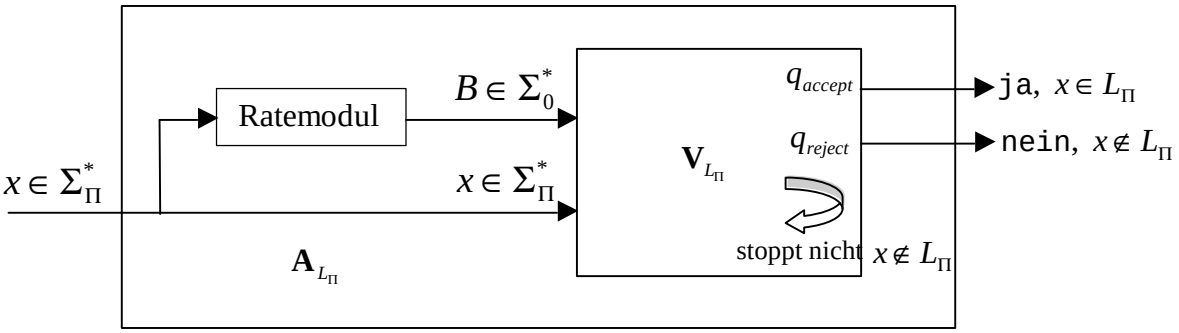
Eingabe: $x \in \Sigma_\Pi^*$, $B \in \Sigma_0^*$

Ausgabe: `ja` (accept), falls $x \in L_\Pi$ gilt
`nein` (reject), falls $x \notin L_\Pi$ gilt.
falls $\mathbf{V}_{L_\Pi}$ bei einer Eingabe $x \in \Sigma_\Pi^*$ nicht hält, gilt ebenfalls $x \notin L_\Pi$.




Falls $\mathbf{V}_{L_\Pi}$ bei Eingabe von $x \in \Sigma_\Pi^*$ stoppt, wird mit $\mathbf{V}_{L_\Pi}(x,B)$, $\mathbf{V}_{L_\Pi}(x,B) \in \{\texttt{ja}, \texttt{nein}\}$, die Entscheidung von $\mathbf{V}_{L_\Pi}$ bezeichnet.

Ein **nichtdeterministischer Algorithmus $\mathbf{A}_{L_\Pi}$ für ein Entscheidungsproblem** $\Pi$ hat die Form




Es gilt also für $x \in \Sigma_\Pi^*$:

$x \in L_\Pi$, falls es einen Beweis $B_x \in \Sigma_0^*$ gibt, so daß $\mathbf{V}_{L_\Pi}$ bei Eingabe von $x$ und $B_x$ mit $\mathbf{V}_{L_\Pi}(x, B_x) = \texttt{ja}$ stoppt;

$x \notin L_\Pi$, falls für jeden Beweis $B \in \Sigma_0^*$ gilt: entweder stoppt $\mathbf{V}_{L_\Pi}$ bei Eingabe von $x$ und $B$ nicht, oder $\mathbf{V}_{L_\Pi}$ stoppt mit $\mathbf{V}_{L_\Pi}(x, B) = \texttt{nein}$.

Beispiel:

**Erfüllbarkeit Boolescher Ausdrücke**

Nicht jeder Boolesche Ausdruck, selbst wenn er syntaktisch korrekt ist, ist erfüllbar. Mit Hilfe eines deterministischen Algorithmus kann man bespielsweise die Erfüllbarkeit eines Booleschen Ausdrucks $F$ mit $n$ Variablen dadurch testen, daß man systematisch nacheinander alle $2^n$ möglichen Belegungen der Variablen in $F$ mit Wahrheitswerten `TRUE` bzw. `FALSE` erzeugt. Immer, wenn eine neue Belegung generiert worden ist, wird diese in die Variablen eingesetzt und der Boolesche Ausdruck ausgewertet. Die Entscheidung, ob $F$ erfüllbar ist oder nicht, kann u.U. erst nach Überprüfung aller $2^n$ Belegungen erfolgen. Das Verfahren ist daher von der Ordnung $O(2^n)$.

Ein nichtdeterministischer Algorithmus würde bei Eingabe eines Booleschen Ausdrucks $F$ mit $n$ Variablen (in geeigneter Kodierung) in Phase 1 zunächst nichtdeterministisch eine 0-1-Folge der Länge $n$ erzeugen, die einer Belegung der $n$ Variablen entspricht. In Phase 2 wird dann diese Belegung in $F$ eingesetzt, $F$ ausgewertet und die Entscheidung „$F$ ist erfüllbar“ genau dann getroffen, wenn bei dieser Auswertung der Wahrheitswert `TRUE` entsteht. In Phase 1 kann, falls $F$ erfüllbar ist, in der Tat eine erfüllende Belegung der Variablen von $F$ er-

zeugt werden. Das geht nicht, wenn *F* nicht erfüllbar ist; in diesem Fall kann in Phase 1 eine beliebige 0-1-Folge erzeugt werden, die, eingesetzt in die Variablen von *F*, stets den Wahrheitswert FALSE ergibt.

Ein Verifizierer zur Überprüfung der Erfüllbarkeit eines Booleschen Ausdrucks *F* mit *n* Variablen erhält als Eingabe den Ausdruck *F* in geeigneter Kodierung und als Beweis $B_F$ eine 0-1-Folge der Länge *n*, die als Belegung der Variablen in *F* interpretiert wird. Der Verifizierer setzt die Belegung in die Variablen von *F* ein. Falls *F* mit $B_F$ erfüllbar ist, trifft der Verifizierer die Entscheidung `ja`, sonst `nein`.

Der Verifizierer für das Erfüllbarkeitsproblem können so entworfen werden, daß er ein Laufzeitverhalten der Ordnung $O(n)$ hat. Insgesamt ist die Laufzeit dieses nichtdeterministischen Algorithmus also von der Ordnung $O(n)$. Es ist nicht bekannt, ob es einen deterministischen Algorithmus gibt, der die Erfüllbarkeit Boolescher Ausdrücke mit einem Laufzeitverhalten der Ordnung $O(p(n))$ mit einem Polynom *p* testet.

# 3 Grenzen der Berechenbarkeit

In Kapitel 2 werden zwei Ansätze vorgestellt, um Berechenbarkeit zu modellieren. Beide Modelle haben sich als äquivalent erwiesen in dem Sinn, daß die Fähigkeit, etwas zu berechnen, beiden Modellen in gleicher Weise zukommt. Selbst das Konzept des Nichtdeterminismus erweitert nicht die Berechenbarkeit, da nichtdeterministisches Verhalten deterministisch simuliert werden kann, auch wenn dabei die Dauer einer Berechnung exponentiell wächst. Von den Modellen, die in Kapitel 2 behandelt werden, zeichnet sich das Modell der Turingmaschine aufgrund seiner Einfachheit und Universalität und nicht zuletzt aus historischen Gründen gegenüber den anderen Modellen aus.

Im folgenden wird unter dem Begriff Turingmaschine eine *k*-DTM verstanden (siehe Kapitel 2.1) mit einer dem jeweiligen Problem angemessenen geeigneten Anzahl *k* an Bändern.

Es sei $\Sigma$ ein endliches Alphabet und $L \subseteq \Sigma^*$. Die Menge $L$ heißt **rekursiv aufzählbar**, wenn es eine Turingmaschine *TM* gibt mit $L = L(TM)$.

Die Bezeichnung *rekursiv aufzählbar* leitet sich aus der Tatsache her, daß eine von einer Turingmaschine akzeptierte Menge $L \subseteq \Sigma^*$ durch eine totale (d.h. überall definierte) berechenbare Funktion „aufgezählt“ werden kann. Genauer:

**Satz 3-1:**

$L \subseteq \Sigma^*$ mit $L \neq \varnothing$ ist genau dann rekursiv aufzählbar, wenn es eine totale berechenbare Funktion $h : (\Sigma \cup \{0,1,\#\})^* \to \Sigma^*$ gibt mit $L = h\left((\Sigma \cup \{0,1,\#\})^*\right)$.

Die Funktion $h$ „zählt $L$ rekursiv auf“ (erzeugt $L$).

**Beweis:**

Diese Aussage enthält zwei „Richtungen“, deren Beweise beide eine genauere Betrachtung verdienen:

Die eine „Richtung“ dieser Aussage, nämlich der Nachweis der Existenz einer totalen und berechenbaren Funktion $h : (\Sigma \cup \{0,1,\#\})^* \to \Sigma^*$ mit $L = h\left((\Sigma \cup \{0,1,\#\})^*\right)$, d.h. einer Funktion *h*, die *L* aufzählt, wird folgendermaßen bewiesen. Wegen $L \neq \varnothing$ gibt es ein Wort $w_0 \in L$. Da *L* als rekursiv aufzählbar angenommen wird, gibt es eine Turingmaschine *TM* mit $L = L(TM)$. Es wird eine Turingmaschine *TM'* angegeben, die zwei Bänder mehr als *TM*

besitzt (nämlich ein zusätzliches Eingabeband und ein Ausgabeband), bei jeder Eingabe stoppt und die gesuchte Funktion $h:(\Sigma\cup\{0,1,\#\})^*\to\Sigma^*$ berechnet:

Bei Eingabe von $w\in(\Sigma\cup\{0,1,\#\})^*$ prüft *TM'* zunächst, ob *w* die Form $w=u\#bin(i)$ mit $u\in\Sigma^*$ und der 0-1-Folge $bin(i)$ hat. Falls nicht, schreibt *TM'* das Wort $w_0$ auf sein Ausgabeband und stoppt. Falls *w* diese Form hat, schreibt *TM'* den Teil *u* auf das 1. Band von *TM* und simuliert das Verhalten von *TM* für höchstens *i* viele Schritte. Falls *TM* das Wort *u* innerhalb dieser Schrittzahl akzeptiert, wird *u* auf das Ausgabeband geschrieben, und *TM'* stoppt. Falls *TM* das Wort *u* innerhalb dieser Schrittzahl nicht akzeptiert, schreibt *TM'* das Wort $w_0$ auf sein Ausgabeband und stoppt. Es sei *h* die von *TM'* berechnete Funktion. Dann gilt $L=h\left((\Sigma\cup\{0,1,\#\})^*\right)$:

Ist nämlich $u\in L$, dann akzeptiert *TM* das Wort in einer endlichen Anzahl *i* von Schritten. Mit $w=u\#bin(i)$ gilt $h(w)=u$. Das bedeutet $L\subseteq h\left((\Sigma\cup\{0,1,\#\})^*\right)$.

Ist umgekehrt $u\in h\left((\Sigma\cup\{0,1,\#\})^*\right)$, etwa $u=h(w)$ für ein Wort $w\in(\Sigma\cup\{0,1,\#\})^*$, dann ist entweder $u=w_0$ oder $w=u\#bin(i)$ und *TM'* hat höchstens *i* viele Schritte von *TM* simuliert und dabei $u\in L$ festgestellt. In beiden Fällen ist also $u\in L$, d.h. $h\left((\Sigma\cup\{0,1,\#\})^*\right)\subseteq L$.

Zum Beweis der anderen „Richtung“ der Aussage wird angenommen, daß $L=h\left((\Sigma\cup\{0,1,\#\})^*\right)$ mit einer totalen und berechenbaren Funktion *h* gilt, und es ist zu zeigen, daß es eine Turingmaschine *TM* gibt, die genau *L* akzeptiert:

Bei Eingabe von $w\in\Sigma^*$ verhält sich *TM* wie folgt. *TM* erzeugt alle Wörter $u\in(\Sigma\cup\{0,1,\#\})^*$ in lexikographischer Reihenfolge. Sobald ein Wort *u* erzeugt ist, berechnet *TM* den Wert $h(u)$. Gilt $h(u)=w$, so stoppt *TM* und akzeptiert *w*. Andernfalls wird das nächste Wort *u* erzeugt. Es läßt sich $L=L(TM)$ zeigen:

Ist $w\in L$, dann ist nach Voraussetzung $w=h(u)$ für ein Wort $u\in(\Sigma\cup\{0,1,\#\})^*$. Da $|u|<\infty$ ist, findet *TM* dieses Wort *u* (bei der Erzeugung aller Wörter in lexikographischer Reihenfolge. Da *h* total und berechenbar ist, kann *TM* den Wert $h(u)$ ermitteln und feststellen, daß $h(u)=w$ gilt. Daher wird *w* von *TM* akzeptiert, d.h. $L\subseteq L(TM)$.

Ist $w\notin L$, dann gilt für jedes $u\in(\Sigma\cup\{0,1,\#\})^*$: $h(u)\neq w$. Dann stoppt *TM* bei Eingabe von *w* nicht, d.h. $w\notin L(TM)$. Daher gilt $L(TM)\subseteq L$. ///

In Kapitel 1.1 wurde der Begriff der Abzählbarkeit definiert: Eine Menge *M* ist abzählbar unendlich, wenn es eine bijektive Abbildung $f:\mathbf{N}\to M$ gibt, d.h. $M=\{f(i)\,|\,i\in\mathbf{N}\}$, und jedes Element $m\in M$ trägt eine eindeutige Nummer *i*: $m=f(i)$. Die Elemente von *M* können mit natürlichen Zahlen durchnumeriert bzw. indiziert werden. Ersetzt man in dieser Definition den Begriff „bijektiv“ durch „total und berechenbar“ und **N** durch $(\Sigma\cup\{0,1,\#\})^*$, so kommt man auf den Begriff der rekursiven Aufzählbarkeit. Für eine rekursiv aufzählbare Menge *L*

gilt $L = h\left(\left(\Sigma \cup \{0, 1, \#\}\right)^*\right)$ mit einer totalen und berechenbaren Funktion $h : \left(\Sigma \cup \{0, 1, \#\}\right)^* \to \Sigma^*$. Die Elemente $u \in L$ können aus Elementen gewonnen werden, die als Zusatzinformation eine obere Schranke für die Schrittzahl enthalten, die eine Turingmaschine benötigt, um das jeweilige Wort zu akzeptieren.

Die Buchstaben des endlichen Alphabets $\Sigma = \{a_0, .., a_{|\Sigma|-1}\}$ einer Turingmaschine lassen sich als Zeichenkette über dem Alphabet {0, 1} kodieren: der Buchstabe $a_i$ hat dabei die Kodierung $bin(a_i)$. Ein Wort $w = a_1 \ldots a_n$ kann man dann zunächst in $bin(a_1)\# \ldots \# bin(a_n)\#$ umsetzen und dann die darin vorkommenden einzelnen Zeichen aus {0, 1, #} in eine 0-1-Folge, wie es etwa in Kapitel 2.4 beschrieben wurde, umkodieren. Entsprechend kann man Paare $(w, v)$ von Worten $w = a_1 \ldots a_n$ und $v = b_1 \ldots b_m$ zunächst in $(bin(a_1)\# \ldots \# bin(a_n)\#\# bin(b_1)\# \ldots \# bin(b_m)\#)$ umsetzen und die darin vorkommenden Zeichen aus { 0, 1, #, (, ) } ähnlich wie in Kapitel 2.4 in eine 0-1-Folge umkodieren. Je nach Anwendung soll es daher erlaubt sein, daß eine Turingmaschine Eingaben der Form $w \in \Sigma^*$, aber auch Eingaben der Form $(w, v) \in \Sigma^* \times \Sigma^*$ verarbeitet. Letztlich lassen sich diese aus Paaren bestehenden Eingaben mit „normalen“ 0-1-Folgen identifizieren.

Der obige Satz kann dann auch so formuliert werden:

**Satz 3-2:**

$L \subseteq \Sigma^*$ mit $L \neq \varnothing$ ist genau dann rekursiv aufzählbar, wenn es eine totale berechenbare Funktion $h : \{0, 1\}^* \to \Sigma^*$ gibt mit $L = h\left(\{0, 1\}^*\right)$.

Die Funktion *h* „zählt *L* rekursiv auf“ (erzeugt *L*).

Im folgenden wird die Klasse der Sprachen, die von Turingmaschinen erkannt werden, d.h. die Klasse der rekursiv aufzählbaren Sprachen, genauer beschrieben. Es ist klar, daß jede rekursiv aufzählbare Menge über einem Alphabet $\Sigma$ endlich oder abzählbar unendlich ist:

**Satz 3-3:**

Die Klasse der rekursiv aufzählbaren Mengen über einem Alphabet $\Sigma$ ist in der Klasse abzählbaren Teilmengen von $\Sigma^*$ enthalten.

Es fragt sich, ob umgekehrt jede abzählbare Teilmenge von $\Sigma^*$ bereits rekursiv aufzählbar ist. Folgende Überlegungen zeigen, daß diese Frage verneint werden muß.

Zu jeder rekursiv aufzählbaren Teilmenge von $\Sigma^*$ gibt es nach Definition eine Turingmaschine, die die Menge akzeptiert. Natürlich kann eine rekursiv aufzählbare Menge von verschiedenen Turingmaschinen akzeptiert werden. Andererseits ist jede von einer Turingmaschine akzeptierte Menge rekursiv aufzählbar. Daher gibt es höchstens so viele rekursiv aufzählbare Teilmengen von $\Sigma^*$ wie Turingmaschinen mit dem Alphabet $\Sigma$. Die Menge der Turingmaschinen mit dem Alphabet $\Sigma$ ist abzählbar; denn jede Turingmaschine *TM* läßt sich mit Hilfe der injektiven Funktion *code* auf ein endlich langes Wort *code*(*TM*) über $\{0,1\}$ abbilden, siehe Kapitel 2.4). Satz 1.1-4 sagt aus, daß es überabzählbar viele Teilmengen von $\Sigma^*$ gibt. Daher existieren auch nicht-rekursiv aufzählbare Teilmengen von $\Sigma^*$.

Eine derartige nicht rekursiv aufzählbare Teilmenge von $\Sigma^*$ kann auch konstruktiv durch eine Technik angegeben werden, die man als **Diagonalisierung** bezeichnet. Der Einfachheit halber soll hier $\Sigma = \{0,1\}$ angenommen werden. Die Elemente von $\Sigma^* = \{0,1\}^*$ werden in lexikographischer Reihenfolge aufgezählt:

| Nummer *i* | Wort $w_i \in \{0,1\}^*$ | Nummer *i* | Wort $w_i \in \{0,1\}^*$ |
|---|---|---|---|
| 0 | *e* | 11 | 100 |
| 1 | 0 | 12 | 101 |
| 2 | 1 | 13 | 110 |
| 3 | 00 | 14 | 111 |
| 4 | 01 | 15 | 0000 |
| 5 | 10 | 16 | 0001 |
| 6 | 11 | 17 | 0010 |
| 7 | 000 | 18 | 0011 |
| 8 | 001 | 19 | 0100 |
| 9 | 010 | 20 | 0101 |
| 10 | 011 | ... | ... |

Das *i*-te Wort $w_i$ kann als Eingabe für eine Turingmaschine mit Eingabealphabet {0, 1} und auch, falls $VERIFIZIERE\_TM(w_i) = \mathsf{TRUE}$ gilt, als die von $w_i$ kodierte Turingmaschine $K_{w_i}$ interpretiert werden (siehe Kapitel 2.4). Es läßt sich daher eine Matrix *M* definieren, deren Zeilen und Spalten mit den Wörtern $w_i$ markiert sind. Die Zeilenmarkierung $w_i$ stellt ein derartiges Eingabewort für eine Turingmaschine dar, die Spaltenmarkierung $w_j$ bezeichnet eventuell die Turingmaschine $K_{w_j}$. Im Schnittpunkt der *i*-ten Zeile von *M* mit der *j*-ten Spalte wird entweder der Wert 0 oder der Wert 1 eingetragen, und zwar so, daß dort

$$\begin{cases} 0 & \text{für } VERIFIZIERE\_TM(w_j) = \mathsf{FALSE} \text{ oder } w_i \notin L(K_{w_j}) \\ 1 & \text{für } VERIFIZIERE\_TM(w_j) = \mathsf{TRUE} \text{ und } w_i \in L(K_{w_j}) \end{cases}$$

steht. Da die ersten Spaltenmarkierungen $w_0$, $w_1$, $w_2$, ... nicht sinnvolle Turingmaschinen kodieren, enthält die Matrix *M* im linken Teil nur die Einträge 0.

**Satz 3-4:**

Die Menge Menge $L_d \subseteq \{0,1\}^*$ sei definiert durch:

Es ist $w \in L_d$ genau dann, wenn $w = w_i$ ist (das *i*-te Wort in der lexikographischen Reihenfolge) und *M* in der *i*-ten Zeile und *i*-ten Spalte den Eintrag 0 hat.

Dann gilt:

Die so definierte Menge $L_d$ ist nicht rekursiv aufzählbar, d.h. es gibt keine Turingmaschine *TM* mit $L(TM) = L_d$.

**Beweis:**

Offensichtlich ist $L_d \neq \emptyset$.

Angenommen, $L_d$ ist rekursiv aufzählbar. Dann gibt es eine Turingmaschine *TM* mit $L(TM) = L_d$. Die Kodierung dieser Turingmaschine sei $w_i$, d.h. $TM = K_{w_i}$ und $L_d = L(K_{w_i})$. Gilt nun $w_i \in L_d$? Falls $w_i \in L_d$ gilt, dann enthält nach Definition von $L_d$ die Matrix *M* in der *i*-ten Zeile und *i*-ten Spalte den Eintrag 0. Nach Definition von *M* und wegen $VERIFIZIERE\_TM(w_i) = \textbf{TRUE}$ bedeutet das aber: $w_i \notin L(K_{w_i})$, also $w_i \notin L_d$. Dieser Widerspruch erlaubt nur die Alternative $w_i \notin L_d$. Nach Definition von $L_d$ enthält die Matrix *M* in der *i*-ten Zeile und *i*-ten Spalte den Eintrag 1. Das bedeutet nach Definition von *M*: $w_i \in L(K_{w_i})$ und damit $w_i \in L_d$. Dieser Widerspruch zeigt, daß die ursprüngliche Annahme, daß $L_d$ rekursiv aufzählbar sei, falsch ist. ///

## 3.1 Eigenschaften rekursiv aufzählbarer und rekursiver Mengen

In Kapitel 2.1 wurden bereits einige Eigenschaften rekursiv aufzählbarer Mengen angegeben:

**Satz 3.1-1:**

1. Sind $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Sigma^*$ rekursiv aufzählbar, dann sind auch $L_1 \cup L_2$ und $L_1 \cap L_2$ rekursiv aufzählbar. Die Klasse der rekursiv aufzählbaren Mengen über einem endlichen Alphabet $\Sigma$ ist abgeschlossen gegenüber Vereinigung- und Schnittbildung.
2. Ist $L \subseteq \Sigma^*$ rekursiv aufzählbar, dann ist nicht notwendigerweise auch $\Sigma^* \setminus L$ rekursiv aufzählbar. Die Klasse der rekursiv aufzählbaren Mengen über einem endlichen Alphabet $\Sigma$ ist nicht abgeschlossen gegenüber Komplementbildung.

Die Gültigkeit der 2. Aussage ergibt sich aus den folgenden Sätzen 3.1-4 und 3.1-6.

Die Menge $L \subseteq \Sigma^*$ werde von der *k*-DTM *TM* akzeptiert. Wird $w \in L$ auf das Eingabeband von *TM* gegeben, dann hält *TM* nach endlich vielen Schritten im Zustand $q_{accept}$ an. Wird ein Wort *w* mit $w \in \Sigma^* \setminus L$ auf das Eingabeband gegeben, dann hält *TM* eventuell nicht an. Es wäre natürlich wünschenswert, daß *TM* auch in diesem Fall anhält, z.B. im Zustand $q_{reject}$, der ausdrückt, daß $w \notin L$ gilt. Gibt es also zu jeder von einer Turingmaschine *TM* akzeptierten Sprache $L \subseteq \Sigma^*$ eine Turingmaschine $TM'$ mit $L(TM') = L$ und der Eigenschaft, daß $TM'$ bei jedem Wort $w \in \Sigma^*$ anhält: im Zustand $q_{accept}$, falls $w \in L$ ist, und im Zustand $q_{reject}$, falls $w \notin L$ ist (nichtstoppende Berechnungen kommen nicht vor)? Diese Frage führt auf die folgende Definition:

Die Menge $L \subseteq \Sigma^*$ heißt **entscheidbar (**oder **rekursiv)**, wenn es eine Turingmaschine *TM* gibt mit der Eigenschaft:
*TM* stoppt bei jeder Eingabe $w \in \Sigma^*$, und *TM* akzeptiert *w* genau dann, wenn $w \in L$ ist.

In praktischen Anwendungen sind die entscheidbaren Mengen gerade diejenigen Sprachen, mit denen man „umgehen“ kann. Denn man möchte ja bei Eingabe von $w \in \Sigma^*$ in einen Entscheidungsalgorithmus nach endlicher Zeit die Frage geklärt haben, ob *w* eine spezifizierte Eigenschaft hat, d.h. ob $w \in L$ ist , oder nicht. Ist *L* entscheidbar, läßt sich diese Entscheidung definitiv herbeiführen. Bei einer rekursiv aufzählbaren Menge *L* und einer passenden Turingmaschine (Entscheidungsverfahren), der man eine Eingabe $w \in \Sigma^*$ vorgelegt hat und die bereits einige Rechenschritte vollzogen hat, ohne bisher eine Entscheidung zu treffen, kann man nicht sicher sein, ob überhaupt jemals eine Entscheidung getroffen wird (vgl. dazu Satz 3.1-4).

**Satz 3.1-2:**

1. Jede entscheidbare Menge ist rekursiv aufzählbar.
   Die Klasse der entscheidbaren Mengen über einem endlichen Alphabet $\Sigma$ ist in der Klasse der rekursiv aufzählbaren Mengen über diesem Alphabet enthalten.

2. Jede endliche Menge ist entscheidbar.

3. Ist $L \subseteq \Sigma^*$ entscheidbar, dann ist auch $\Sigma^* \setminus L$ entscheidbar.
   Die Klasse der entscheidbaren Mengen über einem endlichen Alphabet $\Sigma$ ist abgeschlossen gegenüber Komplementbildung.

4. Sind $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Sigma^*$ entscheidbar, dann sind auch $L_1 \cup L_2$ und $L_1 \cap L_2$ entscheidbar.
   Die Klasse der entscheidbaren Mengen über einem endlichen Alphabet $\Sigma$ ist abgeschlossen gegenüber Vereinigung- und Schnittbildung.

5. Es sei $f : \Sigma^* \to \Sigma'^*$ eine totale und von einer Turingmaschine berechenbare Funktion. Ist $L' \subseteq \Sigma'^*$ entscheidbar, dann ist auch $\overset{-1}{f}(L')$ entscheidbar.

6. Es sei $h : \Sigma^* \to \Sigma'^*$ eine totale und von einer Turingmaschine berechenbare Funktion. Für die Sprachen $L \subseteq \Sigma^*$ und $L' \subseteq \Sigma'^*$ gelte die Eigenschaft
   $w \in L \Leftrightarrow h(w) \in L'$

   Ist $L'$ entscheidbar, dann ist auch $L$ entscheidbar.
   Ist $L$ nicht entscheidbar, dann ist auch $L'$ nicht entscheidbar.

   Ist $L'$ rekursiv aufzählbar, dann ist auch $L$ rekursiv aufzählbar.

   Bemerkung: Man sagt in diesem Fall, daß $L$ auf $L'$ **reduzierbar** sei und schreibt dafür $L \leq L'$.

**Beweis:**

Zu 1. Die Aussage ergibt sich direkt aus der Definition.

Zu 2. Ist $L = \{w_1, ..., w_n\}$ eine endliche Menge, $L \subseteq \Sigma^*$, so ist eine auf allen Eingaben $w \in \Sigma^*$ stoppen Turingmaschine *TM* anzugeben, die genau dann im akzeptierenden Zustand hält, wenn $w$ mit einem der endlich vielen Werte in $L$ übereinstimmt. Es ist also eine Turingmaschine zu definieren, die das (Pseudocode-) Statement

```
CASE w OF
w_1 : accept;
...
w_n : accept;
ELSE reject;
END;
```

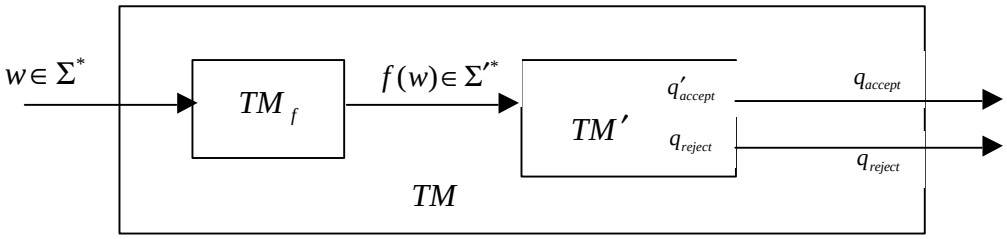
realisert.

Zu 3. Es sei *TM* eine auf allen Eingaben $w \in \Sigma^*$ stoppende Turingmaschine mit $L = L(TM)$. Man kann annehmen, daß *TM* den akzeptierenden Zustand $q_{accept}$ hat, der erreicht wird, wenn $w \in L$ gilt, und den verwerfenden Zustand $q_{reject}$ hat, wenn $w \notin L$ ist. Eine auf allen Eingaben $w \in \Sigma^*$ stoppende Turingmaschine $TM'$ mit $L(TM') = \Sigma^* \setminus L$ erhält man aus *TM*, indem man die Rollen von $q_{accept}$ und $q_{reject}$ vertauscht.

Zu 4. Diese Aussage wird genauso gezeigt wie bei rekursiv aufzählbaren Mengen.

Zu 5. Es sei $TM'$ eine auf allen Eingaben $u \in \Sigma'^*$ stoppende Turingmaschine mit $L(TM') = L'$. Die Funktion $f : \Sigma^* \to \Sigma'^*$ werde durch die Turingmaschine $TM_f$ berechnet. Durch Hintereinanderschalten von $TM_f$ und $TM'$ erhält man eine Turingmaschine *TM*, die auf allen Eingaben $w \in \Sigma^*$ stoppt (weil *f* total ist und $TM'$ auf allen Eingaben stoppt) und für die

$L(TM) = f^{-1}(L') = \{ w \mid f(w) \in L' \}$ gilt:




Zu 6. Aus der Eigenschaft $w \in L \Leftrightarrow h(w) \in L'$ folgt $h^{-1}(L') = L$. Mit 5. ergibt sich die Aussage über die Entscheidbarkeit von *L*. Auf ähnliche Weise wird die Aussage über die rekursive Aufzählbarkeit gezeigt (im vorherigen Bild wird dazu nur der mit $q_{accept}$ markierte Ausgang betrachtet). ///

**Satz 3.1-3:**

Die Klasse der entscheidbaren Mengen ist eine echte Untermenge der Klasse der rekursiv aufzählbaren Mengen.

Dazu ist mindestens ein Beispiel einer rekursiv aufzählbaren Menge, die nicht entscheidbar sind, anzugeben. Dieses kann konstruktiv geschehen:

Zur Erinnerung (siehe Kapitel 2.4): Für ein Wort $w \in \{0,1\}^*$ ist *VERIFIZIERE_TM*(*w*) = TRUE genau dann, wenn *w* die Kodierung einer Turingmaschine ist. Die durch *w* kodierte Turingmaschine wird mit $K_w$ bezeichnet.

Die zum **Halteproblem gehörende Sprache** $L_H \subseteq \{0,1,\#\}^*$ wird definiert als die Menge

$$L_H = \left\{ u\#w \,\middle|\, \begin{array}{l} u \in \{0,1\}^*, w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \text{TRUE und} \\ K_w \text{ stoppt bei Eingabe von } u \end{array} \right\}.$$

Es gilt:

**Satz 3.1-4:**

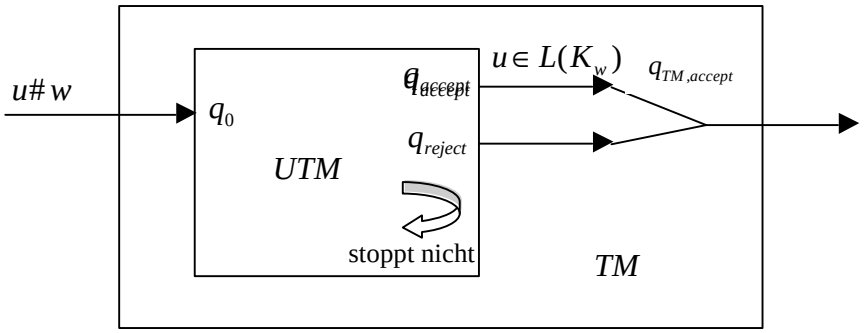
$L_H$ ist rekursiv aufzählbar, aber nicht entscheidbar.

Insbesondere heißt das: es gibt keine immer anhaltende Turingmaschine *TM* mit akzeptierendem Zustand $q_{accept}$ und nicht-akzeptierendem Zustand $q_{reject}$ mit folgender Eigenschaft: Bei Eingabe von *u#w* stoppt *TM* im akzeptierenden Zustand $q_{accept}$, falls $K_w$ auf *u* stoppt, und *TM* stoppt im nicht-akzeptierendem Zustand $q_{reject}$, falls $K_w$ auf *u* nicht stoppt.

**Beweis:**

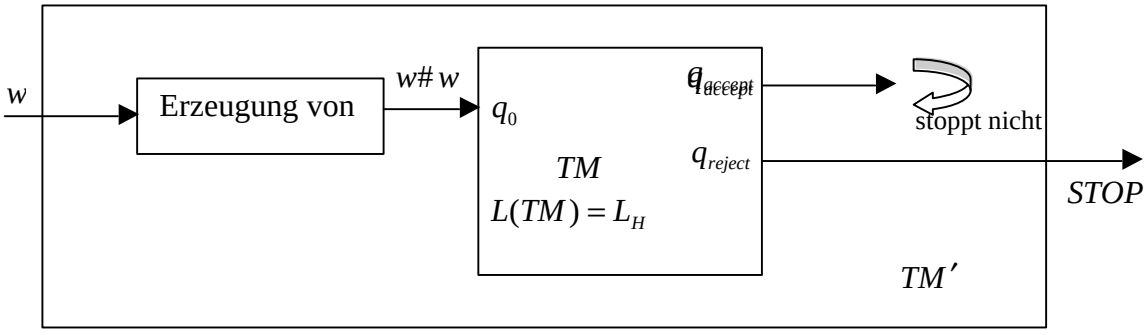
Um die rekursive Aufzählbarkeit von $L_H$ zu zeigen, ist eine Turingmaschine *TM* mit $L(TM) = L_H$ anzugeben. *TM* ist eine einfache Modifikation der in Kapitel 2.4 beschriebenen universellen Turingmaschine *UTM*: Man kann annehmen, daß *UTM* nur ein Band hat und einen akzeptierenden Zustand $q_{accept}$ und einen nicht-akzeptierenden Zustand $q_{reject}$ besitzt. Kommt *UTM* in einen dieser beiden Zustände, stoppt *UTM*. Eventuell stoppt *UTM* jedoch bei einer Eingabe nicht. Es wird ein neuer Zustand $q_{TM,accept}$ in *TM* eingeführt und die Überführungsfunktion $d_{UTM}$ von *UTM* so geändert, daß noch eine Überführung in den Zustand $q_{TM,accept}$ ausgeführt wird, wenn *UTM* in $q_{accept}$ oder $q_{reject}$ kommt. Dazu wird $d_{UTM}$ um die Zeilen $d_{UTM}(q_{accept}, a) = (q_{TM,accept}, (a,S))$ und $d_{UTM}(q_{reject}, a) = (q_{TM,accept}, (a,S))$ für jedes

Symbol *a* aus dem Arbeitsalphabet von *UTM* erweitert. Eine Eingabe $u\#w \in \{0,1,\#\}^*$ für *TM* wird in *UTM* eingegeben. Ist $u\#w \in L(UTM)$, d.h. *VERIFIZIERE_TM*(*w*) = **TRUE** und $u \in L(K_w)$, dann stoppt *UTM* im Zustand $q_{accept}$, und *TM* geht in den Zustand und stoppt; ist $u\#w \notin L(UTM)$, weil *UTM* in den Zustand $q_{reject}$ kommt, dann geht *TM* ebenfalls in den Zustand $q_{TM,accept}$ und stoppt. Offensichtlich gilt $L(TM) = L_H$.




Zum Beweis der Nichtentscheidbarkeit von $L_H$ wird wieder die Methode der Diagonalisierung eingesetzt:

Angenommen, es gibt eine auf allen Eingaben $z \in \{0,1,\#\}^*$ stoppende Turingmaschine *TM* mit $L(TM) = L_H$. Dann wird *TM* leicht abgewandelt zu einer Turingmaschine *TM′*, die wie folgt arbeitet: *TM′* liest eine Eingabe $w \in \{0,1\}^*$ und erzeugt mit einer Kopie von *w* das Wort *w#w*. Dann verhält sich *TM′* wie *TM* bei Eingabe von *w#w*. Falls *TM* im akzeptierenden Zustand $q_{accept}$ stoppt, geht *TM′* in einen neuen Zustand $q'$ über und stoppt nicht[3]. Falls *TM* im nicht-akzeptierendem Zustand $q_{reject}$ stoppt, dann stoppt *TM′*; dabei ist es irrelevant, ob *TM′* in einem akzeptierenden oder einem nicht-akzeptierenden Zustand stoppt.




[3] Ist *TM′* eine *k*-DTM mit der Überführungsfunktion $d'$, dann ist
$d'(q', a_1, ..., a_k) = (q', (a_1, S), ..., (a_k, S))$ für $a_1 \in \Sigma, ..., a_k \in \Sigma$.

Falls *TM* existiert, dann auch $TM'$, und zwar mit einer Kodierung $u \in \{0,1\}^*$, d.h. $TM' = K_u$. Nun gibt es zwei Alternativen: $TM'$ stoppt auf der eigenen Kodierung *u*, oder $TM'$ stoppt auf der eigenen Kodierung *u* nicht. Im ersten Fall ist $u\#u \notin L_H$; nach Definition von $L_H$ stoppt $K_u = TM'$ bei Eingabe von *u* nicht. Dieser Widerspruch läßt nur die zweite Alternative zu; das bedeutet aber $u\#u \in L_H$, also stoppt $K_u = TM'$ bei Eingabe von *u*. Beide Alternativen führen auf einen Widerrspruch. Daher existiert *TM* nicht. ///

Die zum **Halteproblem mit leerem Eingabeband gehörende Sprache** $L_{H_0} \subseteq \{0,1\}^*$ wird definiert als die Menge

$L_{H_0} = \{w \mid w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \text{TRUE} \text{ und } K_w \text{ stoppt bei leerem Eingabeband}\}$.

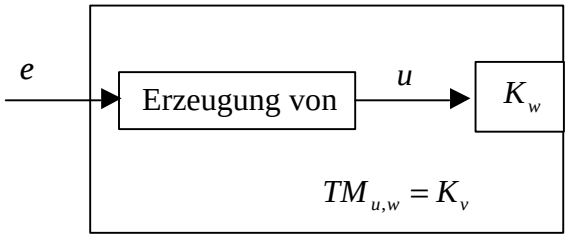
**Satz 3.1-5:**

$L_{H_0}$ ist rekursiv aufzählbar, aber nicht entscheidbar.

**Beweis:**

Man zeigt, daß $L_H$ nach $L_{H_0}$ reduzierbar ist, d.h. daß $L_H \leq L_{H_0}$ gilt. Dazu ist eine totale und berechenbare Funktion $h: \{0,1,\#\}^* \to \{0,1\}^*$ anzugeben, für die $z \in L_H$ genau dann gilt, wenn $h(z) \in L_{H_0}$ ist. Aus Satz 3.1-2, Teil 6 folgt dann die Aussage des Satzes.

Zu $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ mit *VERIFIZIERE_TM*(*w*) = TRUE sei $TM_{u,w}$ eine Turingmaschine, die folgendes Verhalten aufweist: $TM_{u,w}$ startet mit leerem Eingabeband und erzeugt das Wort $u \in \{0,1\}^*$. Dann simuliert $TM_{u,w}$ das Verhalten von $K_w$. Die Kodierung von $TM_{u,w}$ sei *v*. Es gilt $v \in \{0,1\}^*$ und *VERIFIZIERE_TM*(*v*) = TRUE.




Mit Hilfe eines entsprechenden Algorithmus, eines „Turingmaschinengenerators“, kann man bei Vorgabe von *u* und *w* die Kodierung *v* von $TM_{u,w}$ auf deterministische Weise erzeugen. Die Funktion *h* wird definiert durch

$$h: \begin{cases} \{0, 1, \#\}^* & \rightarrow \{0,1\}^* \\ & \\ z & \rightarrow \begin{cases} code(TM_{u,w}) & \text{falls } z = u\#w \text{ mit } u \in \{0,1\}^* \text{ und } w \in \{0,1\}^* \\ & \text{und } VERIFIZIERE\_TM(w) = \text{TRUE} \\ 1 & \text{sonst} \end{cases} \end{cases}$$

Diese ist total und berechenbar. Nach Definition von $L_H$ gilt:

$u\#w \in L_H \Leftrightarrow K_w$ stoppt bei Eingabe von $u$

$\Leftrightarrow TM_{u,w} = K_v$ stoppt bei leerem Eingabeband

$\Leftrightarrow h(u\#w) = v$ und $v \in L_{H_0}$ . ///

Eine Charakterisierung der Entscheidbarkeit einer Menge liefert der folgende Satz:

**Satz 3.1-6:**

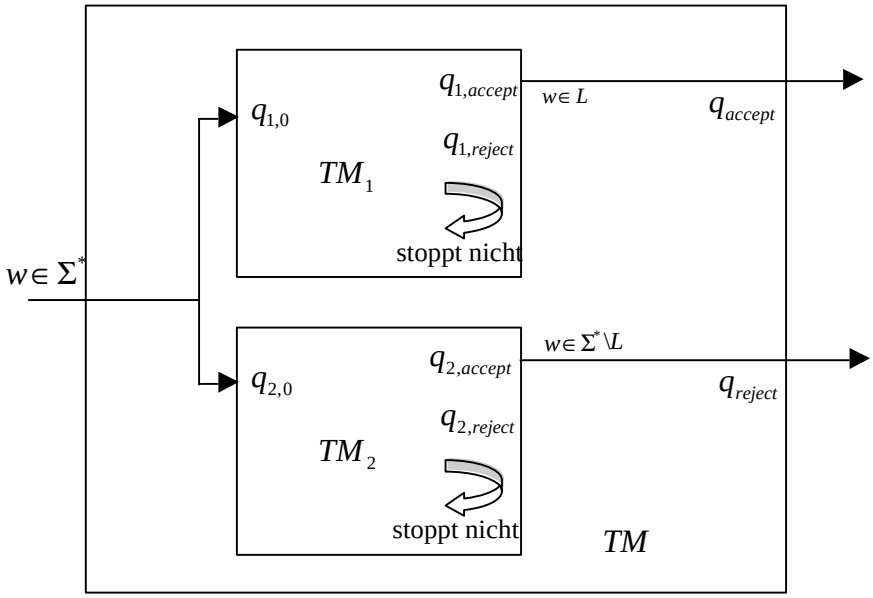
Eine Menge $L \subseteq \Sigma^*$ ist genau dann entscheidbar, wenn sowohl $L$ als auch das Komplement $\Sigma^* \setminus L$ von $L$ rekursiv aufzählbar sind.

**Beweis:**

Auch dieser Satz beinhaltet wieder zwei „Richtungen“:

Ist die Menge $L \subseteq \Sigma^*$ entscheidbar, dann ist sie rekursiv aufzählbar. Satz 3.1-2 Teil 3. besagt, daß mit $L$ auch $\Sigma^* \setminus L$ entscheidbar und damit rekursiv aufzählbar ist.

Gilt umgekehrt für eine Menge $L \subseteq \Sigma^*$, daß mit ihr auch das Komplement $\Sigma^* \setminus L$ rekursiv aufzählbar ist, dann gibt es zwei Turingmaschinen $TM_1$ und $TM_2$ mit $L = L(TM_1)$ und $\Sigma^* \setminus L = L(TM_2)$. Aus diesen beiden Turingmaschinen wird durch Parallelschaltung eine auf allen Eingaben $w \in \Sigma^*$ stoppende Turingmaschine $TM$ konstruiert, die $w$ genau dann akzeptiert, wenn $w \in L$ gilt. Dabei wird eine ähnliche Konstruktion gewählt, wie sie zum Nachweis der Abgeschlossenheit der rekusiv aufzählbaren Mengen bezüglich Vereinigungsbildung angegeben wurde (Kapitel 2.1):




Es gilt für $w \in \Sigma^*$: Ist $w \in L$, dann stoppt $TM_1$ im Zustand $q_{1,accept}$ und damit stoppt *TM* im Zustand $q_{accept}$. Ist $w \notin L$, dann stoppt $TM_2$ im Zustand $q_{2,accept}$ und damit stoppt *TM* im Zustand $q_{reject}$. Da entweder $w \in L$ oder $w \notin L$ gilt, stoppt *TM* bei allen Eingaben $w \in \Sigma^*$ und akzeptiert genau die Menge *L*. ///

Die oben aufgeführte zum Halteproblem gehörige Menge $L_H$ ist rekursiv aufzählbar, aber nicht entscheidbar. Das Komplement von $L_H$, die Menge $\{0,1,\#\}^* \setminus L_H$, kann daher nicht rekursiv aufzählbar sein. Daher ist die Klasse der rekursiv aufzählbaren Mengen gegenüber Komplementbildung nicht abgeschlossen (das ist Satz 3.1-1 Teil 2.).

Da die Rollen von *L* und $\Sigma^* \setminus L$ im letzten Satz „symmetrisch“ sind, ergibt sich als Konsequenz:

Es sei $L \subseteq \Sigma^*$. Dann gilt

**Satz 3.1-7:**

Entweder

1. sowohl *L* als auch $\Sigma^* \setminus L$ ist entscheidbar

oder

2. weder *L* als noch $\Sigma^* \setminus L$ ist entscheidbar

oder

3. entweder *L* oder $\Sigma^* \setminus L$ ist rekursiv aufzählbar, aber nicht entscheidbar, und die jeweils andere Menge ist nicht rekursiv aufzählbar.

Die Klasse der rekursiv aufzählbaren Mengen über einem Alphabet $\Sigma$ ist eine echte Teilklasse von $\mathbf{P}(\Sigma^*)$. Die Klasse der entscheidbaren Mengen über $\Sigma$ ist echt enthalten in der Klasse der rekursiv aufzählbaren Mengen. Es stellt sich die Frage, ob man mit Hilfe eines durch eine Turingmaschine definierten algorithmischen Verfahrens die Klasse der entscheidbaren bzw. die Klasse der nicht-entscheidbaren Mengen erkennen kann oder wenigstens rekursiv aufzählen kann. Genauer: Es wird danach gefragt, ob es eine Turingmaschine gibt, die die Kodierungen aller derjenigen Turingmaschinen erzeugt (rekursiv aufzählt), deren akzeptierte Sprachen gerade die entscheidbaren bzw. nichtentscheidbaren Mengen sind. Leider ist das nicht möglich, wie folgender Satz zeigt:

**Satz 3.1-8:**

Es seien

$L_e = \{w \mid w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } L(K_w) \text{ ist entscheidbar}\}$ und

$L_{ne} = \{w \mid w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } L(K_w) \text{ ist nicht entscheidbar}\}$.

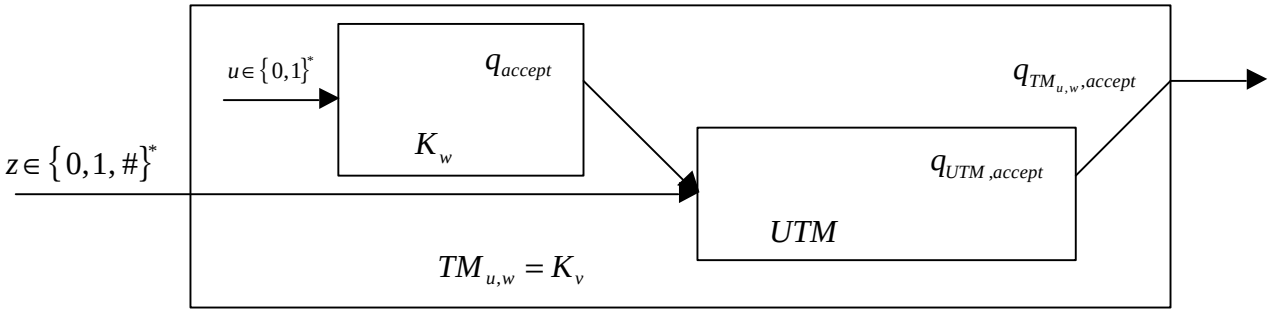
Dann ist weder $L_e$ noch $L_{ne}$ rekursiv aufzählbar (und damit auch nicht entscheidbar).

**Beweis:**

Es wird die von der universellen Turingmaschine *UTM* akzeptierte Sprache $L_{uni} = L(UTM) = \left\{ u\#w \;\middle|\; \begin{array}{l} u \in \{0,1\}^*, w \in \{0,1\}^*, \\ \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } u \in L(K_w) \end{array} \right\} \subseteq \{0,1,\#\}^*$ herangezogen. $L_{uni}$ ist rekursiv aufzählbar, aber nicht entscheidbar. Daher ist das Komplement $\overline{L}_{uni} = \{0,1,\#\}^* \setminus L_{uni}$ nicht rekursiv aufzählbar. Es läßt sich zeigen, daß $\overline{L}_{uni} \leq L_e$ gilt; das Aussehen der dabei beteiligten totalen und berechenbaren Funktion $h : \{0,1,\#\}^* \to \{0,1\}^*$ wird im folgenden skizziert. Aus Satz 3.1-2 Teil 6. folgt, daß $L_e$ nicht rekursiv aufzählbar ist. Der Beweis für $L_{ne}$ verläuft analog.

Ähnlich wie oben wird zu $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ mit *VERIFIZIERE_TM*(*w*) = TRUE mit Hilfe eines deterministisch arbeitenden Turingmaschinengenerators eine Turingmaschine $TM_{u,w}$ konstruiert, die wie folgt arbeitet:

Bei Eingabe von $z \in \{0,1,\#\}^*$ wird die Eingabe zunächst nicht berücksichtigt, sondern $TM_{u,w}$ verhält sich wie $K_w$ auf der Eingabe *u*. Falls $u \in L(K_w)$ festgestellt wird, simuliert $TM_{u,w}$ das Verhalten von *UTM* auf der Eingabe *z*. Die Kodierung von $TM_{u,w}$ sei *v*. Es gilt $v \in \{0,1\}^*$ und *VERIFIZIERE_TM*(*v*) = TRUE. Das Wort *v* ist deterministisch berechenbar, da nur Beschreibungen von Turingmaschinen zusammengestellt wurden.




Es gilt $L(K_v) = L(TM_{u,w}) = \begin{cases} L(UTM) = L_{uni} & \text{falls } u \in L(K_w) \text{ ist} \\ \varnothing & \text{falls } u \notin L(K_w) \text{ ist} \end{cases}$

Bemerkung: Offensichtlich ist $L(K_v)$ genau dann entscheidbar, wenn $u \notin L(K_w)$ ist; denn dann ist $L(K_v) = \varnothing$. Für $u \in L(K_w)$ ist $L(K_v) = L_{uni}$, und diese Menge ist nicht entscheidbar.

Es sei $w_0 \in \{0,1\}^*$ die Kodierung einer Turingmaschine mit $L(K_{w_0}) = \{0,1\}^*$. Es ist $w_0 \in L_e$. Die gesuchte Funktion *h* wird definiert durch

$$h: \begin{cases} \{0,1,\#\}^* \to \{0,1\}^* & \\ z \to \begin{cases} code(TM_{u,w}) & \text{falls } z = u\#w \text{ mit } u \in \{0,1\}^* \text{ und } w \in \{0,1\}^* \\ & \text{und } VERIFIZIERE\_TM(w) = \mathsf{TRUE} \\ w_0 & \text{sonst} \end{cases} \end{cases}$$

Wie obige Ausführungen zeigen, ist *h* ist total und berechenbar. Außerdem gilt

$z \in \overline{L}_{uni} \Leftrightarrow$ *z* hat nicht die Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$
oder *VERIFIZIERE_TM*(*w*) = FALSE
oder $z \notin L_{uni}$, d.h. $u \notin L(K_w)$.

In den ersten beiden Fällen ist $h(z) = w_0$, also $h(z) \in L_e$. Im dritten Fall ist $h(z) = code(TM_{u,w}) = v$ und nach obiger Bemerkung und nach Definition von $L_e$: $h(z) \in L_e$, insgesamt also $z \in \overline{L}_{uni} \Leftrightarrow h(z) \in L_e$, und damit ist $\overline{L}_{uni} \le L_e$ gezeigt. ///

Ähnlich verhält es sich mit den totalen berechenbaren Funktionen. Für ein Wort $w \in \{0,1\}^*$ sei $f_{K_w}: \{0,1\}^* \to \{0,1\}^*$ die von der Turingmaschine $K_w$ berechnete Funktion. Dann gilt:

**Satz 3.1-9:**

Die Menge $L_t = \{ w \mid f_{K_w} \text{ ist eine totale Funktion} \}$ ist nicht rekursiv aufzählbar.

Im folgenden wird nach einem allgemeinen **algorithmischen Verfahren** gefragt, das einer Turingmaschine irgendeine nichttriviale Eigenschaft ansieht. Unter einem algorithmischen Verfahren ist hierbei eine auf allen Eingaben (entweder akzeptierend oder nicht akzeptierend) stoppende Turingmaschine zu verstehen. Der Begriff „nichttriviale Eigenschaft“ wird wie folgt definiert:

Eine Menge $L \subseteq \{0,1\}^*$ von Kodierungen von Turingmaschinen heißt **nichttriviales Entscheidungsproblem über Turingmaschinen**, wenn gilt:

(i) $L \neq \emptyset$

(ii) $L$ enthält nicht die Kodierungen aller Turingmaschinen

(iii) für zwei Turingmaschinen $TM_1$ und $TM_2$, die durch $w_1$ bzw. $w_2$ kodiert werden, d.h. $TM_1 = K_{w_1}$ und $TM_2 = K_{w_2}$, impliziert $L(TM_1) = L(TM_2)$: Es gilt $w_1 \in L$ genau dann, wenn $w_2 \in L$ gilt.

Die **Frage nach der Entscheidbarkeit eines nichttrivialen Entscheidungsproblems** $L$ **über Turingmaschinen** lautet dann in unterschiedlichen äquivalenten Formulierungen:

- Ist $L$ entscheidbar?
- Gibt es ein algorithmisches Verfahren, das bei Eingabe eines Wortes $w \in \{0,1\}^*$ nach endlich vielen Schritten entscheidet, ob $w \in L$ gilt oder nicht?
- Gibt es ein algorithmisches Verfahren, das bei Eingabe der Beschreibung einer Turingmaschine nach endlich vielen Schritten entscheidet, ob die Kodierung der Turingmaschine zu $L$ gehört oder nicht?
- Die Turingmaschinen, deren Kodierungen zu $L$ gehören, haben alle eine durch $L$ beschriebene Eigenschaft $E_L$. **Gibt es ein algorithmisches Verfahren, das bei Eingabe einer Turingmaschine nach endlich vielen Schritten entscheidet, ob die Turingmaschine die Eigenschaft $E_L$ aufweist oder nicht?**
- Kann man mit Hilfe eines algorithmischen Verfahrens entscheiden, ob eine Turingmaschine eine definierte Eigenschaft $E_L$ aufweist oder nicht?

Beispiele nichttrivialer Entscheidungsprobleme:

1. $L = \{ w \mid$ die durch $w$ kodierte Turingmaschine $K_w$ berechnet eine konstante Funktion $\}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob eine Turingmaschine eine konstante Funktion berechnet?
2. Es sei $g$ eine Turingberechenbare Funktion.
   $L = \{ w \mid$ die durch $w$ kodierte Turingmaschine $K_w$ berechnet eine mit $g$ identische Funktion $\}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine berechnete Funktion mit $g$ übereinstimmt?
3. $L_{=\varnothing} = \{ w \mid$ für die durch $w$ kodierte Turingmaschine $K_w$ gilt $L(K_w) = \varnothing \}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine akzeptierte Menge leer ist?
4. $L_{\neq\varnothing} = \{ w \mid$ für die durch $w$ kodierte Turingmaschine $K_w$ gilt $L(K_w) \neq \varnothing \}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine akzeptierte Menge mindestens ein Wort enthält?
5. $L = \{ w \mid$ für die durch $w$ kodierte Turingmaschine $K_w$ gilt : $L(K_w)$ ist endlich $\}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine akzeptierte Menge endlich ist?
6. Es sei $L_0 \subseteq \{0,1\}^*$ eine entscheidbare Menge und
   $L = \{ w \mid$ für die durch $w$ kodierte Turingmaschine $K_w$ gilt : $L(K_w) = L_0 \}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob eine Turingmaschine die Menge $L_0$ akzeptiert?

**Satz 3.1-10:**

Jedes nichttriviale Entscheidungsproblem $L \subseteq \{0,1\}^*$ über Turingmaschinen ist nicht entscheidbar.

**Beweis:**

Es wird gezeigt, daß entweder $L_{H_0} \leq L$ oder $L_{H_0} \leq \{0,1\}^* \setminus L$ gilt. Da $L_{H_0}$ nicht entscheidbar ist, ist im ersten Fall $L$ nicht entscheidbar (Satz 3.1-2 Teil 6.), im zweiten Fall ist $\{0,1\}^* \setminus L$ nicht entscheidbar und damit auch $L$ nicht (Satz 3.1-2 Teil 3.).

Der Beweis folgt dem gleichen Schema wie die Beweise der Sätze 3.1-5 und 3.1-8:

Es sei $TM_0$ eine Turingmaschine mit $L(TM_0) = \varnothing$ und $w_0 = code(TM_0)$, d.h. $TM_0 = K_{w_0}$. Es werden die beiden Fälle

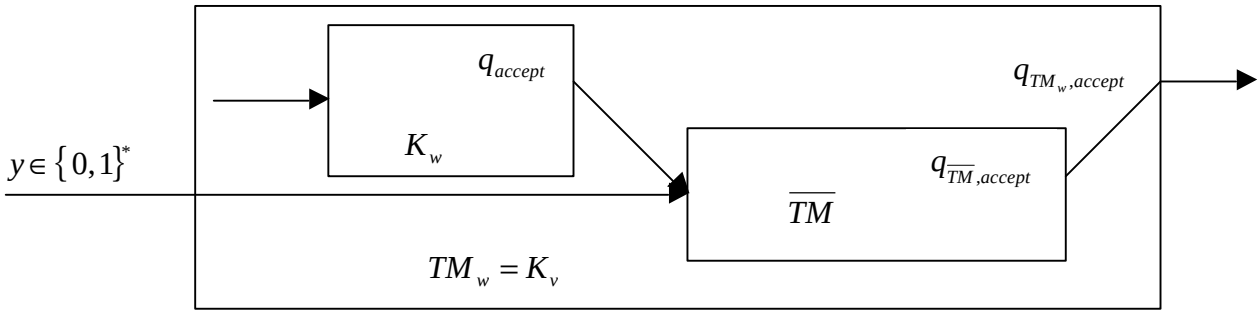
1. Fall: $w_0 \in L$ und

2. Fall: $w_0 \notin L$
unterschieden.

Zum 1. Fall ( $w_0 \in L$ ):

Nach obiger Bedingung (ii) gibt es eine Turingmaschine $\overline{TM}$ mit Kodierung $\overline{w} \in \{0,1\}^*$ und $\overline{w} \notin L$. Es wird die Gültigkeit von $L_{H_0} \leq \{0,1\}^* \setminus L$ gezeigt, indem eine totale und berechenbare Funktion $h : \{0,1\}^* \to \{0,1\}^*$ angegeben wird, für die $h \in L_{H_0} \Leftrightarrow h \in \{0,1\}^* \setminus L \Leftrightarrow h \notin L$ gilt.

Es sei $w \in \{0,1\}^*$ mit *VERIFIZIERE_TM*(*w*) = TRUE. Mit Hilfe eines deterministisch arbeitenden Turingmaschinengenerators wird aus *w* eine Turingmaschine $TM_w$ konstruiert, die wie folgt arbeitet:
Bei Eingabe von $y \in \{0,1\}^*$ wird die Eingabe zunächst nicht berücksichtigt, sondern $TM_w$ simuliert das Verhalten von $K_w$ bei leerem Eingabeband. Falls $K_w$ stoppt, wird *y* in $\overline{TM}$ eingegeben, und $TM_w$ simuliert das Verhalten von $\overline{TM}$ auf der Eingabe *y*. Die Kodierung von $TM_w$ sei *v*. Es gilt $v \in \{0,1\}^*$ und *VERIFIZIERE_TM*(*v*) = TRUE. Das Wort *v* ist deterministisch berechenbar, da nur Beschreibungen von Turingmaschinen zusammengestellt wurden.




Man sieht:

$$L(K_v) = L(TM_w) = \begin{cases} L(\overline{TM}) & \text{falls } w \in L_{H_0} \text{ ist} \\ \varnothing & \text{sonst} \end{cases}.$$

Die Funktion *h* wird definiert durch:

$$h : \begin{cases} \{0,1\}^* & \to & \{0,1\}^* \\ w & \to & \begin{cases} code(TM_w) & \text{falls } VERIFIZIERE\_TM(w) = \text{TRUE} \\ w_0 & \text{sonst} \end{cases} \end{cases}$$

Die Funktion *h* ist total und berechenbar. Außerdem ist diese Funktion für den Nachweis der Relation $L_{H_0} \leq \{0,1\}^* \setminus L$ geeignet:

Ist $w \in L_{H_0}$, dann ist (nach Definition von $L_{H_0}$) *VERIFIZERE_TM*(*w*) = TRUE und $h(w) = code(TM_w) = v$. Außerdem gilt (siehe oben) $L(K_v) = L(\overline{TM}) = L(K_{\overline{w}})$. Nach Bedingung (iii) ist wegen $\overline{w} \notin L$ auch $v \notin L$, d.h. $h(w) \notin L$ bzw. $h(w) \in \{0,1\}^* \backslash L$.
Ist $w \notin L_{H_0}$ und ist *VERIFIZIERE_TM*(*w*) = FALSE, dann ist $h(w) = w_0$ und $h(w) \in L$ bzw. $h(w) \notin \{0,1\}^* \backslash L$.
Ist $w \notin L_{H_0}$ und ist *VERIFIZIERE_TM*(*w*) = TRUE, dann ist $h(w) = code(TM_w) = v$ und $L(K_v) = \varnothing = L(K_{w_0})$. Bedingung (iii), jetzt jedoch zusammen mit der Annahme $w_0 \in L$, impliziert $h(w) \in L$ bzw. $h(w) \notin \{0,1\}^* \backslash L$.

Zum 2. Fall ($w_0 \notin L$):
Nach obiger Bedingung (ii) gibt es eine Turingmaschine $\overline{\overline{TM}}$ mit Kodierung $\overline{\overline{w}} \in \{0,1\}^*$ und $\overline{\overline{w}} \in L$. Es wird die Gültigkeit von $L_{H_0} \leq L$ gezeigt, indem eine totale und berechenbare Funktion $h: \{0,1\}^* \to \{0,1\}^*$ angegeben wird, für die $h \in L_{H_0} \Leftrightarrow h \in L$ gilt. Der Beweis erfolgt wie im 1. Fall; dabei wird die dortige Rolle von $\overline{TM}$ von $\overline{\overline{TM}}$ übernommen. ///

Satz 3.1-10 läßt sich auch so formulieren:

**Satz 3.1-11:**
Ist $L \subseteq \{0,1\}^*$ eine Menge von Kodierungen von Turingmaschinen. Dann ist *L* nur in den Fällen entscheidbar, daß $L = \varnothing$ ist oder daß *L* aus der Menge der Kodierungen aller Turingmaschinen besteht.

Sämtliche oben angeführten Entscheidungsprobleme sind nicht entscheidbar, d.h. es ist beispielsweise nicht entscheidbar, ob

- eine Turingmaschine eine konstante Funktion berechnet
- eine Turingmaschine eine vorgegebene Funktion berechnet
- die von einer Turingmaschine akzeptierte Sprache leer ist
- die von einer Turingmaschine akzeptierte Sprache endlich ist
- die von einer Turingmaschine akzeptierte Sprache mit einer entscheidbaren Menge übereinstimmt.

Der letzte Punkt kann in der Praxis folgendermaßen verwendet werden. Es zeigt sich nämlich, daß Programmverifikation bzw. die Verifikation einer Spezifikation algorithmisch unmöglich ist.

**Satz 3.1-12:**

Für kein algorithmisch lösbares Problem (für keine entscheidbare Menge) läßt sich durch einen einzigen Algorithmus testen, ob ein entworfener Algorithmus eine korrekte Lösung des Problems liefert.

Die Bedingungen, unter denen Mengen aus Kodierungen von Turingmaschinen rekursiv aufzählbar ist, sind wesentlich komplexer:

**Satz 3.1-13:**

Es sei $L \subseteq \{0,1\}^*$ eine Menge von Kodierungen von Turingmaschinen. Dann ist $L$ genau dann rekursiv aufzählbar, wenn folgende Eigenschaften (i) – (iii) gelten:

(i) Ist $w_1 \in L$, $L_1 = L(K_{w_1})$ und ist $L_2$ eine rekursiv aufzählbare Menge, etwa $L_2 = L(K_{w_2})$ für ein Wort $w_2 \in \{0,1\}^*$, mit $L_1 \subseteq L_2$, dann ist auch $w_2 \in L$.

(ii) Ist $w_1 \in L$ und $L_1 = L(K_{w_1})$ eine unendliche Menge, dann gibt es eine endliche Teilmenge $L_2 \subseteq L_1$ mit $L_2 = L(K_{w_2})$ und $w_2 \in L$.

(iii) Es sei $E \subseteq L$ die Menge der Kodierungen, die Turingmaschinen kodieren, deren akzeptierte Sprachen endlich sind; dann ist $E$ entscheidbar.

Beispielsweise folgt aus Satz 3.1-10, daß die Mengen

$L_{\neq\varnothing} = \{w \mid w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } L(K_w) \neq \varnothing\}$ und

$L_{=\varnothing} = \{w \mid w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } L(K_w) = \varnothing\}$

nicht entscheidbar sind. Es wird jedoch nichts darüber gesagt, ob sie rekursiv aufzählbar sind oder nicht. Aus Satz 3.1.-13 folgt, daß $L_{=\varnothing}$ nicht rekursiv aufzählbar ist; denn Bedingung (i) ist verletzt.

In diesem Fall kann man auch anders argumentieren: Man zeigt direkt, daß die Sprache $L_{\neq\varnothing}$ rekursiv aufzählbar ist und folglich $L_{=\varnothing}$ nicht rekursiv aufzählbar ist (denn sonst wäre nach Satz 3.1-6 $L_{=\varnothing}$ entscheidbar). Dazu ist eine Turingmaschine *TM* anzugeben mit $L(TM) = L_{\neq\varnothing}$.

*TM* arbeitet wie folgt: Als Eingabe erhält *TM* ein Wort $w \in \{0,1\}^*$. Falls *VERIFIZIERE_TM*(*w*) = FALSE ist, wird *w* nicht akzeptiert. Andernfalls beginnt *TM*, alle Worte $z \in \{0,1,\#\}^*$ der Form $z = v\#bin(i)$ mit $v \in \{0,1\}^*$ in lexikographischer Reihenfolge zu erzeugen. Jedesmal, wenn ein derartiges Wort generiert worden ist, simuliert *TM* das Ver-

halten von $K_w$ bei Eingabe von *v* für höchstens *i* Schritte. Wird *v* dabei akzeptiert, akzeptiert *TM* die Eingabe *w*. Andernfalls wird das nächste Wort $v'bin(i')$ erzeugt und getestet. Es gilt:
Ist $w \in L(TM)$, dann gibt es ein Wort $v \in \{0,1\}^*$, so daß $K_w$ dieses Wort in höchstens *i* Schritten für ein $i \in \mathbf{N}$ akzeptiert. Daher ist $L(K_w) \neq \emptyset$ und $w \in L_{\neq\emptyset}$.
Ist umgekehrt $w \in L_{\neq\emptyset}$, d.h. $L(K_w) \neq \emptyset$, dann gibt es ein Wort $v \in L(K_w)$. Dieses Wort wird in endlich vielen, etwa *j* Schritten akzeptiert. Wenn *TM* also bei Eingabe von *w* das Wort $v\#bin(j)$ generiert hat, stoppt $K_w$ nach der Simulation von *j* Schritten im akzeptierenden Zustand, und *TM* akzeptiert *w*, d.h. $w \in L(TM)$.

Für die in Satz 3.1-8 untersuchten Sprachen
$L_e = \{w \mid w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und } L(K_w) \text{ ist entscheidbar}\}$ und
$L_{ne} = \{w \mid w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und } L(K_w) \text{ ist nicht entscheidbar}\}$
folgt ebenfalls aus Satz 3.1-13, daß sie nicht rekursiv aufzählbar sind; in beiden Fällen ist wieder Bedingung (i) verletzt.

Zusammenfassung wichtiger Beispiele:

| nicht entscheidbar, aber rekursiv aufzählbar | nicht rekursiv aufzählbar |
|---|---|
| $L_H = \left\{ u\#w \,\middle|\, \begin{array}{l} u \in \{0,1\}^*, w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ K_w \text{ stoppt bei Eingabe von } u \end{array} \right\}$ | |
| $L_{H_0} = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ K_w \text{ stoppt bei leerem Eingabeband} \end{array} \right\}$ | |
| $\overline{L}_d = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^*, w = w_i \text{ ist das } i\text{-te Wort} \\ \text{in der lexikographischen Ordnung} \\ \text{aller Worte aus } \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ w_i \in L(K_{w_i}) \end{array} \right\}$ | $L_d = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^*, w = w_i \text{ ist das } i\text{-te Wort} \\ \text{in der lexikographischen Ordnung} \\ \text{aller Worte aus } \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{FALSE} \\ \text{oder } w_i \notin L(K_{w_i}) \end{array} \right\}$ |
| $L_{uni} = \left\{ u\#w \,\middle|\, \begin{array}{l} u \in \{0,1\}^*, w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ u \in L(K_w) \end{array} \right\}$ | $\overline{L}_{uni} = \left\{ u\#w \,\middle|\, \begin{array}{l} u \in \{0,1\}^*, w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ u \notin L(K_w) \end{array} \right\}$ |
| | |

../..

| | |
|---|---|
| | |
| $L_{\neq\varnothing} = \left\{ w \middle\| \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ L(K_w) \neq \varnothing \end{array} \right\}$ | $L_{=\varnothing} = \left\{ w \middle\| \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ L(K_w) = \varnothing \end{array} \right\}$ |
| | $L_e = \left\{ w \middle\| \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ L(K_w) \text{ ist entscheidbar} \end{array} \right\}$ |
| | $L_{ne} = \left\{ w \middle\| \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ L(K_w) \text{ ist nicht entscheidbar} \end{array} \right\}$ |
| | $L_t = \left\{ w \middle\| \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ f_{K_w} \text{ ist eine totale Funktion} \end{array} \right\}$ |

# 4 Elemente der Theorie Formaler Sprachen und der Automatentheorie

Bisher wurden Sprachen über die Akzeptanz durch Turingmaschinen definiert. Das Ergebnis ist die Klasse der rekursiv aufzählbaren Sprachen. Diese soll nun weiter strukturiert werden, indem das Modell der Turingmaschine eingeschränkt wird. Eine Einschränkung wurde bereits in Kapitel 3.1 behandelt: es werden dort Turingmaschinen betrachtet, die bei jeder Eingabe stoppen. Diese Spezialisierung führte auf die Klasse der entscheidbaren Sprachen, die eine echte Teilklasse der Klasse der rekursiv aufzählbaren Sprachen ist.

Insgesamt werden in den folgenden Unterkapiteln vier Sprachklassen beschrieben. Diese Einteilung ist nach Noam Chomsky benannt, der diese Klassen als mögliche Modelle für natürliche Sprachen charakterisiert hat. Allerdings definiert die **Chomskyhierarchie** Sprachen nicht über die Akzeptanz durch geeignete Maschinenmodelle, sondern definiert jede Klasse durch die Form von syntaktischen Regeln, nach denen Wörter der jeweiligen Sprache „erzeugt“ werden können. Für jede Sprache wird eine entsprechende (formale) Grammatik festgelegt. Je nach Art der erzeugenden Regeln ist die erzeugte Sprache eine Typ-0-Sprache, Typ-1-Sprache, Typ-2-Sprache bzw. Typ-3-Sprache. Die zugrundeliegende Theorie heißt **Theorie der Formalen Sprachen**. Zu jedem Sprachtypen der Chomskyhierarchie gibt es ein entsprechendes Berechnungsmodell (Automatentyp), das sich aus dem Modell der Turingmaschine (durch die erwähnten Einschränkungen) ableitet. Somit besteht ein enger Zusammenhang zwischen der Theorie der Formalen Sprachen und der **Automatentheorie**, die sich wesentlich mit der Definition von Modellen der Berechenbarkeit und Übersetzbarkeit beschäftigt. Beide Ansätze werden in den folgenden Unterkapiteln gegenübergestellt.

## 4.1 Grammatiken und formale Sprachen

Die Akzeptanz (das Erkennen) einer Sprache $L \subseteq \Sigma^*$ über einem endlichen Alphabet $\Sigma$ mit Hilfe eines Berechnungsmodells wie der Turingmaschine kann als „analytischer“ Ansatz bezeichnet werden. Dem gegenüber steht ein „synthetischer“ Ansatz, der beschreibt, wie die Wörter einer Sprache mit Hilfe von Regeln erzeugt werden können. Dieser Ansatz wird in der **Theorie der formalen Sprachen** verfolgt.

Eine **Grammatik** $G = (\Sigma, N, S, R)$ wird definiert durch

1. das endliche Alphabet $\Sigma$ der **Terminalsymbole**
2. das endliche Alphabet $N$ der **Nichtterminalsymbole (Variablen)**
3. das **Startsymbol** $S \in N$
4. eine endliche Menge $R$ von **Erzeugungsregeln** (**Ableitungsregeln**, **Produktionen**) mit

$$R \subseteq \left\{ v \to w \mid v \in (N \cup \Sigma)^+, w \in (N \cup \Sigma)^*, |v| \geq 1 \right\}.$$

Für $x \in (N \cup \Sigma)^*$, $y \in (N \cup \Sigma)^*$, $v \in (N \cup \Sigma)^*$, $w \in (N \cup \Sigma)^*$ heißt das Wort *xwy* **von** *G* **in einem Schritt aus** *xvy* **erzeugt**, wenn es eine Erzeugungsregel $v \to w$ in *R* gibt. Man schreibt dann: $xvy \overset{v \to w}{\Rightarrow} xwy$ oder $xvy \underset{G}{\Rightarrow} xwy$.

Wenn der Zusammenhang klar ist, werden die Super- bzw. Subskripte auch weggelassen.

Ein Wort $w \in (N \cup \Sigma)^*$ heißt **von** *G* **aus** $x \in (N \cup \Sigma)^*$ **erzeugt**, wenn es eine Folge von Wörtern $x_i \in (N \cup \Sigma)^*$, $i = 0, ..., t$, gibt mit $x = x_0$, $x_i \underset{G}{\Rightarrow} x_{i+1}$ für $i = 0, ..., t-1$, $w = x_t$. Man schreibt dann auch $x \overset{*}{\underset{G}{\Rightarrow}} w$. Die Menge $L(G) = \left\{ w \,\middle|\, w \in \Sigma^* \text{ und } S \overset{*}{\underset{G}{\Rightarrow}} w \right\}$ heißt **die von** *G* **erzeugte Sprache**.

Beispiele:

**Grammatik für einige Sprachen über** $\{a,b,c\}$

1. $G_1 = (\{a,b\}, \{S, A, B\}, S, \{S \to AB, A \to aA, A \to e, B \to bB, B \to e\})$ erzeugt die Sprache $L_1 = \{a^i b^j \mid i \in \mathbf{N},\ j \in \mathbf{N}\}$. Die Sprache $L_1$ wird auch von der Grammatik $G_1' = (\{a,b\}, \{S, B\}, S, \{S \to e, S \to aS, S \to bB, B \to bB, B \to e\})$ erzeugt.

2. $G_2 = (\{a,b\}, \{S\}, S, \{S \to e, S \to aSb\})$ erzeugt die Sprache $L_2 = \{a^n b^n \mid n \in \mathbf{N}\}$

3. $G_3 = (\{a,b\}, \{S, A, B\}, S, \{S \to AB, A \to e, A \to aAb, B \to e, B \to cB\})$ erzeugt die Sprache $L_3 = \{a^n b^n c^i \mid n \in \mathbf{N}, i \in \mathbf{N}\}$

4. $G_4 = (\{a,b,c\}, \{S, B\}, S, \{S \to aSBc, S \to abc, cB \to Bc, bB \to bb\})$ erzeugt die Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1\}$. Die Sprache $L_4$ wird auch von der Grammatik $G_4' = (\{a,b,c\}, \{S, B, C\}, S, \{S \to aSBC, S \to aBC, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc\})$ erzeugt.

5. $G_5 = (\{0,1\}, \{S, A, B\}, S, \{S \to 1A, S \to 0B, A \to 0, A \to 0S, A \to 1AA, B \to 1, B \to 1S, B \to 0BB\})$ erzeugt die Sprache

$L_5 = \{ w \mid w \in \{0,1\}^+ \text{ und die Anzahl der Zeichen 0 in } w \text{ ist gleich der Anzahl der Zeichen 1} \}$

Eine Grammatik $G = (\Sigma, N, S, R)$ kann ähnlich wie eine Turingmaschine (siehe Kapitel 2.4) durch eine Zeichenkette aus $\{0,1\}^*$ (algorithmisch auf deterministische Weise) kodiert werden. Die Kodierung ist dabei eine injektive Abbildung, die jeder Grammatik *G* ein Wort $code(G) \in \{0,1\}^*$ zuordnet. Zusätzlich kann die Codierung so entworfen werden, daß man entscheiden kann, ob ein Wort $w \in \{0,1\}^*$ die Kodierung einer Grammatik darstellt, und daß man aus der Kodierung einer Grammatik diese rekonstruieren kann. Auf Details der Darstellung und der Verfahren soll hier verzichtet werden.

Für ein Wort $w \in \{0,1\}^*$ ist

$$VERIFIZIERE\_G(w) = \begin{cases} \mathsf{TRUE} & \text{falls } w \text{ die Kodierung einer Grammatik darstellt} \\ \mathsf{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Grammatik darstellt} \end{cases}$$

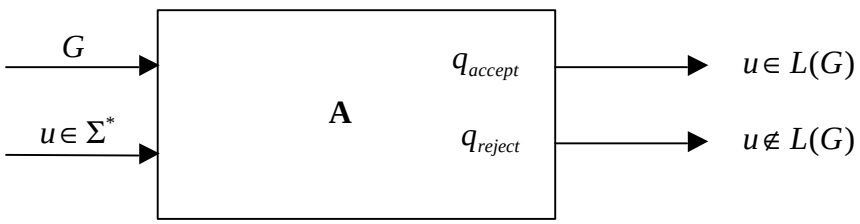
Die durch ein Wort $w \in \{0,1\}^*$ mit *VERIFIZIERE_G*(*w*) = TRUE kodierte Grammatik sei $KG_w$.

Die **zum Wortproblem gehörende Menge** $L_{Wort}$ wird definiert durch

$$L_{Wort} = \left\{ u\#w \;\middle|\; u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE\_G(w) = \mathsf{TRUE} \text{ und } u \in L(KG_w) \right\}.$$

Das **Wortproblem ist entscheidbar**, wenn $L_{Wort}$ entscheidbar ist. In diesem Fall gibt es einen auf allen Eingaben der Form $u\#w$ mit $u \in \Sigma^*$ und $w \in \{0,1\}^*$ stoppenden Algorithmus, der die Eingabe genau dann akzeptiert, wenn $u\#w \in L_{Wort}$ ist.

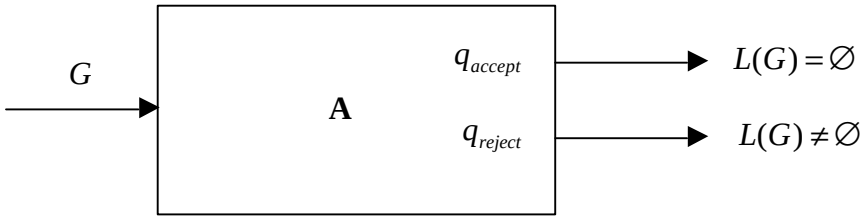
Vereinfacht ausgedrückt ist **das Wortproblem entscheidbar**, wenn es einen auf jeder Eingabe stoppenden Algorithmus **A** gibt, der eine aus zwei Teilen bestehende Eingabe, nämlich bestehend aus (der Kodierung) einer Grammatik $G = (\Sigma, N, S, R)$ und einer Zeichenkette $u \in \Sigma^*$, erhält und die Eingabe genau dann akzeptiert, wenn $u \in L(G)$ gilt, d.h. wenn sich *u* aus dem Startsymbol von *G* durch Anwendung der in *G* definierten Ableitungsregeln herleiten läßt.



Entsprechend kann man das Leerheitsproblem definieren:

Die **zum Leerheitsproblem gehörende Menge** $L_{leer}$ wird definiert durch
$L_{leer} = \{ w \mid w \in \{0,1\}^*, VERIFIZIERE\_G(w) = \mathsf{TRUE} \text{ und } L(KG_w) = \varnothing \}$.

Das **Leerheitsproblem ist entscheidbar** (hier in der vereinfachten Darstellung), wenn es einen auf jeder Eingabe stoppenden Algorithmus **A** gibt, der als Eingabe eine Grammatik $G = (\Sigma, N, S, R)$ erhält und die Eingabe genau dann akzeptiert, wenn $L(G) = \varnothing$ gilt.




## 4.2 Typ-0-Sprachen

Eine Sprache, die von einer Grammatik $G = (\Sigma, N, S, R)$ erzeugt wird, für die $R \subseteq \{ v \to w \mid v \in (N \cup \Sigma)^+ \setminus \Sigma^*, w \in (N \cup \Sigma)^* \}$ gilt, heißt **Typ-0-Sprache**. Die Regeln $v \to w$ einer Typ-0-Sprache erfüllen also die Bedingung $|v| \geq 1$, und die linke Seite *v* einer Regel enthält mindestens ein nichtterminales Symbol; außerdem kann in einer Ableitung durch Anwendung einer Regel keine einmal entstandene rein terminale Zeichenkette mehr ersetzt werden. In einem Ableitungsschritt $xvy \overset{v \to w}{\Rightarrow} xwy$ kann es aber durchaus vorkommen, daß das Ergebnis *xwy* des Ableitungsschritts kürzer als die Ausgangszeichenkette ist.

Man kann zeigen, daß jede Typ-0-Sprache mit einem Alphabet $\Sigma$ von einer Turingmaschine mit Eingabealphabet $\Sigma$ akzeptiert werden kann. Dazu wird der Erzeugungsvorgang einer Grammatik mit Hilfe einer nichtdeterministischen Turingmaschine simuliert. Umgekehrt läßt sich jede rekursiv aufzählbare Menge durch eine Typ-0-Grammatik erzeugen. Daher gilt:

**Satz 4.2-1:**
Die Klasse der rekursiv aufzählbaren Mengen (von Turingmaschinen akzeptierte Mengen) über einem endlichen Alphabet $\Sigma$ ist mit der Klasse der Typ-0-Sprachen mit Alphabet $\Sigma$ identisch.

In Kapitel 3.1 wurde gezeigt, daß die Menge

$$L_{uni} = \left\{ u\#w \,\middle|\, u \in \{0,1\}^*, w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und } u \in L(K_w) \right\}$$

nicht entscheidbar ist. Satz 4.2-1 legt nahe, in der Definition von $L_{uni}$ das Prädikat *VERIFIZIERE_TM* durch *VERIFIZIERE_G* zu ersetzen und dieses dann durch das leicht modifizierte Prädikat

$$VERIFIZIERE\_G\_TYP\_0(w) = \begin{cases} \mathsf{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-0-Grammatik darstellt} \\ \mathsf{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-0-Grammatik darstellt} \end{cases}$$

Man erhält dann das **Wortproblem für Typ-0-Grammatiken**, das danach fragt, ob die Menge

$$L_{Wort\_Typ-0} = \left\{ u\#w \,\middle|\, u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_0(w) = \mathsf{TRUE} \text{ und } u \in L(KG_w) \right\}$$

entscheidbar ist. Diese Frage muß verneint werden.

Ähnlich verhält es sich mit dem **Leerheitsproblem für Typ-0-Grammatiken**. Die Menge

$$L_{leer\_Typ-0} = \left\{ w \,\middle|\, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_0(w) = \mathsf{TRUE} \text{ und } L(KG_w) = \varnothing \right\}$$

ist nicht entscheidbar, da (vgl. Kapitel 3.1) die Menge

$$L_{=\varnothing} = \left\{ w \,\middle|\, w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und } L(K_w) = \varnothing \right\}$$

nicht rekursiv aufzählbar und damit auch nicht entscheidbar ist.

Insgesamt ergibt sich

**Satz 4.2-2:**

Das Wortproblem und das Leerheitsproblem für Typ-0-Grammatiken sind nicht entscheidbar:

Es gibt keinen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-0-Grammatik $G = (\Sigma, N, S, R)$ und eines Wortes $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ ist.

Es gibt keinen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-0-Grammatik $G = (\Sigma, N, S, R)$ entscheidet, ob $L(G) = \varnothing$ gilt.

## 4.3 Typ-1-Sprachen

Es sei $G = (\Sigma, N, S, R)$ eine Grammatik, in der die Erzeugungsregeln

$R \subseteq \{ v \to w \mid v \in (N \cup \Sigma)^+ \setminus \Sigma^*, w \in (N \cup \Sigma)^* \}$ folgender zusätzlicher Einschränkung unterliegen:
Für alle Regeln $v \to w$ gilt $|v| \leq |w|$. Als einzige Ausnahme ist die Regel $S \to e$ zugelassen; wenn diese Regel vorkommt, darf *S* auf keiner rechten Seite einer Regel vorkommen.

Wie bei einer Typ-0-Grammatik dürfen auf der linken Seite einer Regel in einer Typ-1-Grammatik also sowohl terminale als auch nichtterminale Zeichen vorkommen, wobei links mindestens ein nichtterminales Zeichen steht. Die Länge der rechten Seite einer Regel ist bis auf die Ausnahme $S \to e$ mindestens so groß wie die Länge der linken Seite. Daher wird in einem Ableitungsschritt $xvy \overset{v \to w}{\Rightarrow} xwy$ die Länge des Ableitungsergebnisses nicht kürzer. In einer Ableitung $S \overset{*}{\Rightarrow} w$ eines Wortes $w \in \Sigma^+$, etwa $S \Rightarrow x_1 \Rightarrow ... \Rightarrow x_i \Rightarrow x_{i+1} \Rightarrow ... \Rightarrow w$, gilt für alle Zwischenschritte $1 = |S| \leq |x_1| \leq ... \leq |x_i| \leq |x_{i+1}| \leq ... \leq |w|$. Diese Beobachtung wird wichtig, wenn man den Turingmaschinentyp charakterisieren möchte, der eine von einer Typ-1-Grammatik erzeugten Sprache erkennt.

Enthält *R* die Regel $S \to e$, dann ist $e \in L(G)$, und die Anwendung der Regel $S \to e$ stellt die einzige Möglichkeit dar, um $e$ aus *S* abzuleiten.

Die von einer derartigen Grammatik erzeugte Sprache heißt **Typ-1-Sprache** oder **kontextsensitive Sprache**. Die Grammatik heißt **kontextsensitive Grammatik**. Beispielsweise ist die Sprache $L_4 = \{ a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1 \}$ aus Kapitel 4.1 eine kontextsensitive Sprache (Typ-1-Sprache).

Jede Typ-1-Sprache ist natürlich auch eine Typ-0-Sprache.

Es stellt sich die Frage, ob es Typ-0-Sprachen gibt, die nicht kontextsensitiv sind. Zur Beantwortung dieser Frage wird folgender Ansatz gewählt: Es wird ein Berechnungsmodell vorgestellt, das sich aus dem Modell der Turingmaschine ableitet und genau kontextsensitive Sprachen akzeptiert, und dann untersucht, wie sich dieses Berechnungsmodell zum Modell der Turingmaschine verhält:

Ein **linear beschränkter Automat** *LBA* ist eine nichtdeterministische Turingmaschine, deren Schreib/Leseköpfe auf allen Bändern bei Eingabe eines Wortes *w* mit $|w| = n$ jeweils nicht mehr als *n* Zellen auf den Bändern verwenden. Insbesondere bewegt sich kein Kopf weiter nach rechts als bis zur Position $n+1$ (an dem Blank auf dem Eingabeband bei Position $n+1$ wird das Ende des Eingabeworts erkannt). Die von einem *LBA* akzeptierte Menge $L(LBA)$ wird wie bei Turingmaschinen definiert.

Man kann zeigen, daß es zu jeder von einem linear beschränkten Automaten *LBA* akzeptierten Sprache $L = L(LBA)$ eine kontextsensitive Grammatik $G_{LBA}$ gibt mit $L(G_{LBA}) = L(LBA)$. Umgekehrt gibt es zu jeder von einer kontextsensitiven Grammatik *G* erzeugten Sprache $L' = L(G)$ einen linear beschränkten Automaten $LBA_G$ mit $L(LBA_G) = L(G)$. In diese Überlegungen geht wesentlich ein, daß die Produktionen $v \to w$ der kontextsensitiven Grammatik der Bedingung Regeln $|v| \le |w|$ genügen, so daß zur Akzeptanz auf allen Bändern jeweils ein durch die Länge des Eingabeworts beschränkter Speicherplatz ausreicht.

**Satz 4.3-1:**

Die Klasse der von linear beschränkten Automaten akzeptierten Sprachen über einem endlichen Alphabet $\Sigma$ ist mit der Klasse der kontextsensitiven Sprachen (Typ-1-Sprachen) über $\Sigma$ identisch.

Möchte man daher Aussagen über kontextsensitive Sprachen beweisen, so kann man dazu mit kontextsensitiven Grammatiken oder linear beschränkten Automaten argumentieren. Satz 4.3-1 läßt vermuten, daß die Klasse der kontextsensitiven Sprachen über $\Sigma$ eine echte Teilklasse der rekursiv aufzählbaren Sprachen über $\Sigma$ ist. Der folgende Satz bestätigt diese Vermutung.

**Satz 4.3-2:**

Die von einem *LBA* akzeptierte Menge $L(LBA)$ über einem Alphabet $\Sigma$ ist entscheidbar.

Jede kontextsensitive Sprache über einem Alphabet $\Sigma$ ist entscheidbar.

Die Entscheidung kann bei einem Wort $w \in \Sigma^*$ mit Länge *n* in $O(2^{O(n)})$ vielen Schritten getroffen werden; das Entscheidungsverfahren hat also exponentielle Laufzeit.

**Beweis:**

Es sei $G = (\Sigma, N, S, R)$ eine kontextsensitive Grammatik. Zu zeigen ist: $L(G)$ ist entscheidbar. Dazu wird eine auf allen Eingaben $w \in \Sigma^*$ stoppende Turingmaschine $TM_G$ angegeben, die *w* genau dann akzeptiert, wenn $w \in L(G)$ ist. Die Arbeitsweise von $TM_G$ wird hier informell in Form eines Algorithmus beschrieben:

Bei Eingabe von $w \in \Sigma^*$ mit $|w| = n$ erzeugt $TM_G$ einen Graphen, dessen Knoten mit den Zeichenketten in $(N \cup \Sigma)^*$ markiert sind, die eine Länge besitzen, die kleiner oder gleich *n* ist

(andere Zeichenketten kommen in einer Ableitung $S \overset{*}{\Rightarrow} w$ nicht vor). Ist dabei der Knoten $K_i$ mit der Zeichenkette $a \in (N \cup \Sigma)^*$ und der Knoten $K_j$ mit $b \in (N \cup \Sigma)^*$ markiert und kann man in *G* durch Anwendung einer Regel $b$ aus $a$ in einem Ableitungsschritt herleiten, d.h. $a \underset{G}{\Rightarrow} b$, dann wird eine Kante von $K_i$ nach $K_j$ eingefügt. Ein Knoten ist mit *S* markiert und einer mit *w*. Es ist $w \in L(G)$ genau dann, wenn es einen Pfad von dem mit *S* markierten Knoten zu dem mit *w* markierten Knoten gibt. Um diese Tatsache festzustellen, kann man einen der bekannten Algorithmen zum Auffinden von Pfaden in Graphen zwischen definierten Knoten anwenden. Da der beschriebene Graph $O(c^n)$ viele Knoten (mit einer Konstanten $c = c(G)$) besitzt und sich die Laufzeit des Pfadsuchalgorithmus durch eine Funktion der Ordnung $O(m^3)$ beschränken läßt, wobei $m \in O(c^n)$ die Anzahl der Knoten im Graphen angibt, ist das gesamte Entscheidungsverfahren von der Ordnung $O(2^{O(n)})$. ///

Die Klasse der kontextsensitiven Sprachen ist eine echte Teilklasse der Klasse der entscheidbaren Sprachen, wie folgende Sätze zeigen.

Mit der Diagonalisierungstechnik läßt sich zunächst folgender (technischer) Hilfssatz zeigen:

**Satz 4.3-3:**

Es sei

$$L_0 = \left\{ w \,\middle|\, w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } K_w \text{ stoppt auf jeder Eingabe} \right\}$$

eine rekursiv aufzählbare Menge. Dann gibt es eine entscheidbare Sprache *L*, die von einer auf allen Eingaben stoppenden Turingmaschine *TM* erkannt wird, deren Kodierung in $L_0$ nicht vorkommt.

Bemerkung: Da die Menge

$$L_e = \left\{ w \,\middle|\, w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } L(K_w) \text{ ist entscheidbar} \right\}$$

aus Kapitel 3.1 nicht rekursiv aufzählbar ist, gilt $L_0 \neq L_e$.

**Beweis:**

Da $L_0$ als rekursiv aufzählbar angenommen wird, gibt es nach Satz 3-2 eine totale berechenbare Funktion $h : \{0,1\}^* \to \Sigma^*$ mit $L_0 = h(\{0,1\}^*)$.

Es wird $L = \left\{ w \,\middle|\, w \in \{0,1\}^* \text{ und } w \notin L(K_{h(w)}) \right\}$ gesetzt.

Dann ist *L* entscheidbar: Eine auf allen Eingaben $w \in \{0,1\}^*$ stoppende Turingmaschine *TM* mit $L(TM) = L$, d.h. die entscheidet, ob $w \in L$ gilt oder nicht, arbeitet wie folgt: *TM* berech-

net zunächst $h(w)$. Dabei ist $h(w) \in L_0$, insbesondere $VERIFIZIERE\_TM(h(w)) = \mathsf{TRUE}$. Jetzt simuliert *TM* das Verhalten von $K_{h(w)}$ bei Eingabe von *w*. Da $K_{h(w)}$ nach Definition von $L_0$ auf allen Eingaben stoppt, kann *TM* feststellen, ob $w \in L(K_{h(w)})$ gilt oder nicht. Das Wort *w* wird von *TM* genau dann akzeptiert, wenn bei dieser Simulation $w \notin L(K_{h(w)})$ festgestellt wird.

*TM* habe die Kodierung $w_L$. Falls $w_L \in L_0$ gilt, dann sei $w \in \{0,1\}^*$ so gewählt, daß $h(w) = w_L$ ist. Dann folgt (nach Definition von *L*) der Widerspruch

$w \in L \Leftrightarrow w \notin L(K_{h(w)})$ (nach Definition von *L*)

$\Leftrightarrow w \notin L(K_{w_L})$ (wegen $h(w) = w_L$)

$\Leftrightarrow w \notin L(TM)$ (da $w_L$ die Kodierung von *TM* ist)

$\Leftrightarrow w \notin L$ (wegen $L(TM) = L$).

Daher kommt die Kodierung $w_L$ von *TM* in $L_0$ nicht vor. ///

Satz 4.3-3 kann genutzt werden, um zu zeigen, daß es eine entscheidbare Menge gibt, die nicht kontextsensitiv ist:

Jede kontextsensitive Grammatik kann ähnlich wie eine Turingmaschine (siehe Kapitel 2.4 und Kapitel 4) durch eine Zeichenkette aus $\{0,1\}^*$ (algorithmisch auf deterministische Weise) kodiert werden. Wie in Kapitel 2.4 auf der Menge der Turingmaschinen kann man dann auf der Menge der kontextsensitiven Grammatiken auf Basis ihrer Kodierungen eine lineare Ordnung definieren. Man kann also von der *i*-ten kontextsensitiven Grammatik $G_i$ sprechen. Nach Satz 4.3-2 ist $L(G_i)$ entscheidbar mit einer (dort angegebenen) Turingmaschine $TM_{G_i}$; die Turingmaschine $TM_{G_i}$ stoppt (nach Konstruktion) auf jeder Eingabe. Deren Kodierung sei (gemäß dem Vorgehen aus Kapitel 2.4) $code(TM_{G_i})$. Die Berechnung des Werts $code(TM_{G_i})$ bei Vorgabe von *i* bzw. von $bin(i)$ erfolgt insgesamt auf deterministische Weise und definiert eine Funktion $h: \{0,1\}^* \to \{0,1\}^*$:

$(i \leftrightarrow) bin(i) \to i$ - te kontextsensitve Grammatik $G_i \to TM_{G_i} \to code(TM_{G_i})$.

Diese Abbildung ist injektiv, wie man leicht nachprüfen kann. Durch *h* wird jeder Zeichenkette $u = bin(i)$ die Kodierung einer Turingmaschine zugeordnet, die auf jeder Eingabe stoppt. Setzt man $L_0 = h(\{0,1\}^*)$, so sind die Voraussetzungen in Satz 4.3-3 erfüllt. Daher gilt der folgende Satz.

**Satz 4.3-4:**

Die Klasse der kontextsensitiven Sprachen (Typ-1-Sprachen) über einem endlichen Alphabet $\Sigma$ ist eine echte Untermenge der entscheidbaren Sprachen über $\Sigma$ und damit auch der Typ-0-Sprachen über $\Sigma$.

Wie im allgemeinen Fall der Turingmaschine unterscheidet man auch bei linear beschränkten Automaten nichtdeterministisches und deterministisches Verhalten. Ein **nichtdeterministischer linear beschränkter Automat** liegt vor, wenn die Überführungsfunktion die Form $d : Q \times \Sigma^k \to \mathbf{P}\left(Q \times \left(\Sigma \times \{L,R,S\}\right)^k\right)$ hat, ein **deterministischer linear beschränkter Automat** liegt vor, wenn die Überführungsfunktion die Form $d : Q \times \Sigma^k \to Q \times \left(\Sigma \times \{L,R,S\}\right)^k$ hat. Im deterministischen Fall ist die Folgekonfiguration (falls sie überhaupt existiert) einer Konfiguration eindeutig bestimmt; zum Nichtdeterminismus vgl. Kapitel 2.5. Es ist nicht bekannt, ob jede von einem *nichtdeterministischen* linear beschränkten Automaten auch von einem *deterministischen* linear beschränkten Automaten akzeptiert wird. Dieses offene Problem heißt **LBA-Problem**. Zu beachten ist dabei folgendes: Ist $L$ eine kontextsensitive Sprache, dann gibt es einen nichtdeterministischen linear beschränkten Automaten $LBA$ mit $L = L(LBA)$. Die Akzeptanz eines Wortes $w$ mit $|w| = n$ benötigt eine Anzahl von Zellen, die durch die lineare Funktion $S(n) = n + 1$ gegeben ist. Das nichtdeterministische Verhalten einer Turingmaschine, die durch eine Funktion der Ordnung $O(f(n))$ platzbeschränkt ist, kann deterministisch simuliert werden, wobei dabei der Speicherplatzbedarf die Ordnung $O\left(f^2(n)\right)$ annimmt. Da ein linear beschränkter Automat auch eine nichtdeterministische Turingmaschine ist, könnte man versuchen, diese deterministische Simulation hier zu verwenden. Diese führt jedoch aus der Klasse der linear beschränkten Automaten heraus, da sie quadratischen Speicherplatz der Ordnung $O\left(S^2(n)\right) = O\left(n^2\right)$ benötigt. Daher trägt die deterministische Simulation einer nichtdeterministischen Turingmaschine in dieser Allgemeinheit zur Lösung des LBA-Problems nichts bei.

Die Klasse der von deterministischen linear beschränkten Automaten akzeptierten Sprachen ist abgeschlossen gegen Komplementbildung (das zeigt man wie in Satz 3.1-2 Teil 3.). Man hat lange Zeit vermutet, daß diese Abschlußeigenschaft für die Klasse der von nichtdeterministischen linear beschränkten Automaten akzeptierten Sprachen nicht zutrifft. Die Richtigkeit dieser Vermutung hätte die (negative) Lösung des LBA-Problems nach sich gezogen. Inzwischen weiß man jedoch, daß auch die Klasse der von nichtdeterministischen linear beschränkten Automaten akzeptierten Sprachen abgeschlossen gegen Komplementbildung ist. Das LBA-Problem ist weiterhin ungelöst.

Die Definition des Wortproblems für Typ-0-Grammatiken kann man auf Typ-1-Grammatiken übertragen: Dazu wird das Prädikat

$$VERIFIZIERE\_G\_TYP\_1(w)$$

$$= \begin{cases} \mathsf{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-1-Grammatik darstellt} \\ \mathsf{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-1-Grammatik darstellt} \end{cases}$$

definiert. Dieses Prädikat ist berechenbar. Dazu ist unter anderem zu prüfen, ob die Erzeugungsregeln in $KG_w$ den Bedingungen einer kontextsensitiven Grammatik genügen.

Das **Wortproblem für Typ-1-Grammatiken** fragt danach, ob die Menge

$$L_{Wort\_Typ-1} = \left\{ u\#w \,\middle|\, u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_1(w) = \mathsf{TRUE} \text{ und } u \in L(KG_w) \right\}$$

entscheidbar ist.

Aus dem Beweis von Satz 4.3-2 folgt, daß das Wortproblem für kontextsensitive Grammatiken entscheidbar ist. Das **Leerheitsproblem für Typ-1-Grammatiken**, nämlich die Frage, ob die Menge

$$L_{leer\_Typ-1} = \left\{ w \,\middle|\, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_1(w) = \mathsf{TRUE} \text{ und } L(KG_w) = \varnothing \right\}$$

entscheidbar ist, muß wie bei Typ-0-Grammatiken verneint werden (der Beweis findet sich in der angegebenen Literatur).

Zusammenfassend ergibt sich

**Satz 4.3-5:**

Das Wortproblem für Typ-1-Grammatiken ist entscheidbar; das Leerheitsproblem für Typ-1-Grammatiken ist nicht entscheidbar:

Es gibt einen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-1-Grammatik $G = (\Sigma, N, S, R)$ und eines Wortes $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ ist.

Es gibt keinen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-1-Grammatik $G = (\Sigma, N, S, R)$ entscheidet, ob $L(G) = \varnothing$ gilt.

## 4.4 Typ-2-Sprachen

Es sei $G = (\Sigma, N, S, R)$ eine Grammatik, in der für die Erzeugungsregeln $R \subseteq \left\{ A \to w \,|\, A \in N \text{ und } w \in (N \cup \Sigma)^* \right\}$ gilt. Auf der linken Seite einer Regel steht immer genau ein nichtterminales Symbol, rechts kann auch die leere Zeichenkette vorkommen.

Die von einer derartigen Grammatik erzeugte Sprache heißt **Typ-2-Sprache** oder **kontextfreie Sprache**. Die zugehörige Grammatik heißt **kontextfreie Grammatik**. Beispielsweise sind die Sprache $L_2 = \left\{ a^n b^n \,|\, n \in \mathbf{N} \right\}$, $L_3 = \left\{ a^n b^n c^i \,|\, n \in \mathbf{N}, i \in \mathbf{N} \right\}$ und

$L_5 = \{ w \mid w \in \{0,1\}^+$ und die Anzahl der Zeichen 0 in $w$ ist gleich der Anzahl der Zeichen $1\}$ aus Kapitel 4.1 kontextfreie Sprachen (Typ-2-Sprachen).

Die kontextfreien Sprachen zählen zu den am intensivsten erforschten Sprachen. Einen überragenden Erfolg haben sie in der Anwendung und der Theorie des Compilerbaus. Programmiersprachen wie Pascal und ihre Nachfolger sind erst nach intensiver Erforschung der kontextfreien Sprachen und unter Anwendung dieser Theorie entwickelt worden. Die Syntax der meisten heute üblichen Programmiersprachen wird in Form einer Grammatik definiert, die weitestgehend kontextfrei ist. Man kann jedoch formal zeigen, daß eine Programmiersprache, in der Variablen mit Datentypen deklariert werden, so daß Typverträglichkeit verlangt wird, in der die Anzahl von Formal- und Aktualparametern in Prozeduren übereinstimmen müssen, in der ausschließlich die Verwendung vorher deklarierter Objekte zulässig ist usw., nicht komplett durch eine kontextfreie Grammatik beschrieben werden kann.

Beispiel:

**Ausschnitt aus der Sprachdefinition der Programmiersprache Object Pascal**

Die Syntax der Sprache Object Pascal wird wie bei vielen anderen Programmiersprachen (weitgehend) durch eine kontextfreie Grammatik definiert. Nichtterminale Symbole werden dabei durch Bezeichner angegeben, die mit einem Großbuchstaben beginnen und weitere Kleinbuchstaben enthalten. Bezeichner, die nur Großbuchstaben enthalten, stehen für jeweils ein einziges terminales Symbol. So bezeichnet im folgenden Ausschnitt der Bezeichner `Ziel` das Startsymbol der Sprache, während der Bezeichner `UNIT` für ein einziges terminales Symbol steht.

Eine Regelangabe der Form $A \to a_1 \mid a_2 \mid ... \mid a_n$ steht für die $n$ Regeln $A \to a_1$, $A \to a_2$, ..., $A \to a_n$. Ein Angabe in eckigen Klammern innerhalb einer Regel bezeichnet einen optionalen Teil, d.h. $A \to a[b]g$ steht für die Regeln $A \to abg$ und $A \to ag$.

```
Ziel -> (Programm | Package  | Bibliothek  | Unit)
Programm -> [PROGRAM Bezeichner ['('Bezeichnerliste')'] ';']

            Programmblock '.'

Unit -> UNIT Bezeichner ';'

        interface-Abschnitt
        implementation-Abschnitt
        initialization-Abschnitt '.'

Package -> PACKAGE Bezeichner ';'

           [requires-Klausel]
           [contains-Klausel]
```

```
           END '.'

Bibliothek -> LIBRARY Bezeichner ';'

              Programmblock '.'

Programmblock -> [uses-Klausel]

                 Block

uses-Klausel -> USES Bezeichnerliste ';'
interface-Abschnitt -> INTERFACE

                       [uses-Klausel]
                       [interface-Deklaration]...

interface-Deklaration -> const-Abschnitt

                      -> type-Abschnitt
                      -> var-Abschnitt
                      -> exported-Kopf

exported-Kopf -> Prozedurkopf ';' [Direktive]

              -> Funktionskopf ';' [Direktive]

implementation-Abschnitt -> IMPLEMENTATION

                            [uses-Klausel]
                            [Deklarationsabschnitt]...

Block -> [Deklarationsabschnitt]

         Verbundanweisung

Deklarationsabschnitt -> Label-Deklarationsabschnitt

                      -> const-Abschnitt
                      -> type-Abschnitt
                      -> var-Abschnitt
                      -> Prozedurdeklarationsabschnitt

...

Einfache Anweisung -> Designator ['(' Ausdrucksliste ')']

                   -> Designator ':=' Ausdruck
                   -> INHERITED
                   -> GOTO Label-Bezeichner

Strukturierte Anweisung -> Verbundanweisung

                         -> Bedingte Anweisung
                         -> Schleifenanweisung
                         -> with-Anweisung

Verbundanweisung -> BEGIN Anweisungsliste END
Bedingte Anweisung -> if-Anweisung
```