## Die Chomsky-Hierarchie

Durch Verallgemeinerungen der Regeln unserer kontextfreien Grammatiken können wir viel größere Klassen von Sprachen beschreiben. Wir werden also Grammatiken immer noch als $G = \langle V, \Sigma, R, S \rangle$ darstellen, und die generierte Sprache ist wie gewohnt

$L(G) = \{w \in \Sigma^* : S \overset{*}{\underset{G}{\Rightarrow}} w\}$,

nur verlangen wir nicht mehr, dass die linke Seite der Regeln aus einem einzigen Nichtterminalsymbol besteht:

### Typ-0-Grammatiken

Typ-0-Grammatiken haben Regeln

$\alpha \to \beta$, wobei $\alpha \in V^*(V\text{-}\Sigma)V^*$, und $\beta \in V^*$,

Die von Typ-0-Grammatiken generierten Sprachen nennen wir Typ-0-Sprachen oder *rekursiv aufzählbare Sprachen*.

### Typ-1 Grammatiken (kontextsensitive Grammatiken)

sind wie Typ-0-Grammatiken, aber mit der Einschränkung dass $|\alpha| \leq |\beta|$. Es gibt für Typ-1-Grammatiken folgende Normalform:

$\alpha A \gamma \to \alpha \beta \gamma$

(wobei $A$ ein Nichtterminalsymbol ist)

Solche Regeln sind fast wie die kontextfreien Regeln, erlauben aber nur die Substitution von $A$ in einem *Kontext*, daher die Bezeichnung *kontextsensitiv*.

### Typ-2- und Typ-3-Grammatiken

Diese kennen wir schon: Es sind die kontextfreien bzw. regulären Grammatiken.

### Beispiel für eine Typ-0-Grammatik

Die folgende Grammatik $G = \langle V, \Sigma, R, S \rangle$ generiert die Sprache $L(G) = \{a^{2^n} : n \geq 0\}$:

$V = \{[, ], D, S, a\}$

$\Sigma = \{a\}$

$R = \{ S \to [a],$
$\quad [ \to [D,$
$\quad Da \to aaD,$ *(diese Regel ist auch Typ-1)*
$\quad D] \to ],$ *(eine echte Typ-0-Regel)*
$\quad [ \to \varepsilon,$ *(kontextfrei, aber nicht Typ-1)*
$\quad ] \to \varepsilon \}$ *(kontextfrei, aber nicht Typ-1)*

Beliebig viele $D$'s ($D$ für duplizieren) können links eingefügt werden. Sie können nach rechts durch die schon erzeugten $a$'s „wandern", und auf dem Weg werden aus jedem $a$ zwei gemacht. Ein $D$ kann erst dann wieder gelöscht werden, wenn es das ]-Zeichen rechts erreicht hat.

**Beispiel**

Ableitung von *aaaa*:

S ⇒ [a]
⇒ [Da]
⇒ [aaD]
⇒ [aa]
⇒ [Daa]
⇒ [aaDa]
⇒ [aaaaD]
⇒ [aaaa]
⇒ aaaa]
⇒ aaaa

**Theorem**

Die Typ-0-Sprachen bilden eine echte Obermenge der kontextfreien Sprachen.

**Beweis**

Die gerade vorgestellte Sprache $L = \{a^{2^n} : n \geq 0\}$ ist nicht kontextfrei:

Angenommen, $L$ sei kontextfrei. Dann gibt es eine kontextfreie Grammatik $G$ mit $L(G) = L$. Das Pumping-Lemma sagt uns, dass es ein Zahl $K$ gibt, so dass für alle Wörter $w$ mit $|w| > K$ gilt: Es gibt Wörter $u, v, x, y, z$, so dass $w = uvxyz$, $vy \neq \varepsilon$ und $uv^nxy^nz \in L(G)$ für jedes $n \geq 0$.

Sei $w = a^{2^k} = uvxyz \in L(G)$ ein Wort mit $|w| > K$ und $k \geq 0$. Dann muss auch $uv^2xy^2z \in L(G)$ sein. Es gilt: $|w| < |uv^2xy^2z| \leq 2|w|$. Die nächstgrößere Zeichenkette in $L(G)$ nach $w = a^{2^k}$ ist $a^{2^{k+1}}$, und $|a^{2^{k+1}}| = 2\,|w|$. Also muss $u = x = z = \varepsilon$ und $vy = w$ sein. Aber dann ist $|uv^3xy^3z| = 3|w|$. Die nächstgrößere Zeichenkette in $L(G)$ ist aber $a^{2^{k+2}}$, und $|a^{2^{k+2}}| = 4|w|$. Kontradiktion.

Die Sprache ist aber auch mit einer kontextsensitiven Grammatik generierbar (wird nicht gezeigt). Tatsächlich ist es nicht so leicht, Sprachen zu finden, die von keiner kontextsensitiven Grammatik generiert werden, es sei denn man nimmt dann eine kontextfreie Sprache, die $\varepsilon$ enthält. Regeln wie $A \to \varepsilon$ sind nach der Definition einer kontextsensitiven Grammatik nicht zugelassen.

**Theorem**

Die kontextsensitiven Sprachen bilden eine echte Obermenge der ε-freien kontextfreien Sprachen

**Beweis**

Wir betrachten die Sprache $\{a^n b^n c^n : n \geq 1\}$, die ja nicht kontextfrei ist.

Die Sprache wird von einer kontextsensitiven Grammatik mit den folgenden Regeln generiert:

$S \to abc$
$S \to aSBc$
$cB \to Bc$
$bB \to bb$

Z.B. ist *aaabbbccc* in *L(G)*:

$S \Rightarrow aSBc$
$\Rightarrow aaSBcBc$
$\Rightarrow aaabcBcBc$
$\Rightarrow aaabBccBc$
$\Rightarrow aaabbccBc$
$\Rightarrow aaabbcBcc$
$\Rightarrow aaabbBccc$
$\Rightarrow aaabbbccc.$

**Die rekursiven Sprachen**

**Definition**

Eine Sprache $L$ heißt *rekursiv* wenn es einen Algorithmus gibt, der für eine gegebene Zeichenkette $w \in \Sigma^*$ entscheiden kann, ob $w \in L$. Wir wissen schon, dass die kontextfreien Sprachen rekursiv sind. Es gilt auch:

**Theorem**

Jede kontextsensitive Sprache ist rekursiv.

**Beweis**

Sei $G = \langle V, \Sigma, R, S \rangle$ eine kontextsensitive Grammatik und sei $w \in \Sigma^*$ mit $|w| = n$. Sei $V(n)$ die Menge der Wörter über $V$, die höchstens $n$ Zeichen haben. Wir bilden einen Graphen mit den Elementen aus $V(n)$ als Knoten, so dass es von $u$ nach $v$ eine Kante gibt gdw. $v$ von $u$ in $G$ direkt ableitbar ist. Die Pfade im Graphen entsprechen Ableitungen in $G$, und zwar allen möglichen, die eine Zeichenkette der Länge $n$ erzeugen (weil $|\alpha| \leq |\beta|$ wenn $\alpha \Rightarrow \beta$). Also ist $w$ genau dann in $L(G)$, wenn es im Graphen einen Pfad von $S$ nach $w$ gibt. (Wir werden hier aber keinen Pfad-Suche-Algorithmus vorstellen.)

### Korrolar

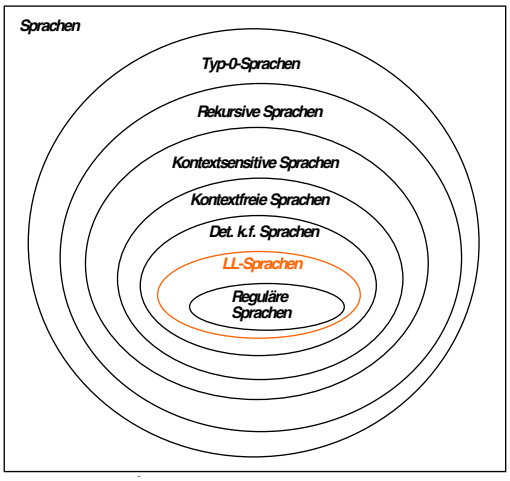
Es gibt Parsingalgorithmen für die kontextsenstiven Grammatiken.

### Beweis

Der im vorigen Beweis vorgestellte Algorithmus erzeugt jede mögliche Ableitung eines Wortes in einer kontextsensitiven Grammatik.

### Übersicht der Sprachklassen

Wir können uns jetzt das folgende Bild der Sprachklassen, die wir kennengelernt haben, bilden:




Wir wissen schon, dass einige dieser Inklusionen echte Inklusionen sind. Es gibt nicht-reguläre deterministisch kontextfreie Sprachen ($\{a^n b^n : n \geq 0\}$), es gibt nicht-deterministische kontextfreie Sprachen (das Komplement von $\{a^n b^n c^n : n \geq 0\}$), und es gibt nicht-kontextfreie kontextsensitive Sprachen ($\{a^n b^n c^n : n \geq 0\}$).

Tatsächlich ist jede der Inklusionen echt – wir lassen aber den Beweis, dass es nicht-kontextsensitive rekursive Sprachen gibt, aus.

Um zu zeigen, dass es nicht-rekursive Typ-0-Sprachen gibt, brauchen wir einen Automaten, der Typ-0-Sprachen akzeptiert, und ein formales Modell eines Algorithmus. Tatsächlich können wir für beide Zwecke denselben Typ von Automaten – die *Turing-Maschinen* – benutzen. Und bis jetzt hat niemand Algorithmen gefunden, die außerhalb des Turing-Maschinen-Modells fallen. Dies gibt eine gute Unterstützung für:

**Die Church'sche Hypothese**

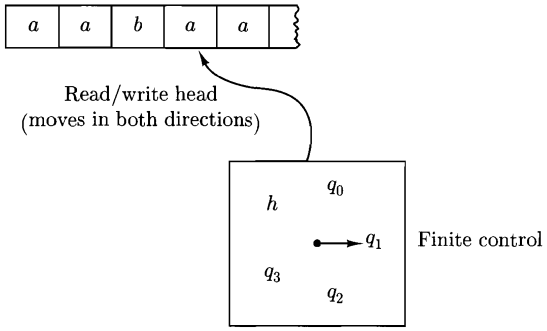
Alles, was berechenbar ist, läßt sich von einer Turing-Maschine berechnen.

Dies ist und bleibt eine Hypothese, solange „berechenbar“ nicht formalisiert werden kann.

Was aber bewiesen werden kann ist, dass andere Algorithmenmodelle mit Turingmaschinen äquivalent sind. Z.B. sind die Typ-0-Grammatiken mit Turingmaschinen äquivalent, und auch die *rekursiven Funktionen,* eine mathematische Formalisierung der Berechenbarkeit.

### Turing-Maschinen

Kurz gesagt hat eine Turing-Maschine wie ein endlicher Automat eine endliche Kontrolle, hat aber ein unendlich langes Eingabeband (ein Teil davon ist die *endliche* Eingabe), auf dem er auch schreiben darf. Außerdem darf sich der Automat auf der Eingabe auch nach *links* bewegen:




Turing-Maschinen brauchen keine Endzustände; wie bei den endlichen Transducern schreibt die Maschine das Ergebnis ihrer Berechnung auf das Band.

Es gibt aber einen besonderen *Haltezustand*, der signalisiert, dass die Berechnung beendet ist. Dieser Zustand ist derselbe für jede Maschine, und wird mit $h$ bezeichnet.

Das Band ist unendlich nach rechts. Wenn die Maschine sich links vom am weitesten links stehenden Zeichen bewegt, hört die Berechnung auf, aber wir sagen dann nicht, dass $M$ hält, sondern dass $M$ *hängt*.

Die Eingabe ist auf einem endlichen Teil des Bandes geschrieben. Der nicht verwendete Teil des Bandes ist mit Leerzeichen (#) gefüllt.

### Turing-Maschinen formal definiert

Eine Turingmaschine $M = \langle K, \Sigma, \delta, s \rangle$ besteht aus:

$K$ – Eine Menge von Zuständen, wobei $h \notin K$

$\Sigma$ – Ein Alphabet, wobei $\# \in \Sigma$ aber $L, R \notin \Sigma$.

$s$ – Ein Startzustand ($s \in K$)

$\delta$ – Eine Funktion von $K \times \Sigma$ in $(K \cup \{h\}) \times (\Sigma \cup \{L, R\})$

$\delta(q, a) = \langle p, b \rangle$ bedeutet, dass die Maschine, wenn sie im Zustand $q$ ist und ein $a$ sieht, in den Zustand $p$ übergehen soll und außerdem:

– wenn $b = L$ oder $b = R$ ist, sich in die entsprechende Richtung auf der Band bewegt

– wenn $b \in \Sigma$ ist, auf der Eingabe das $a$ mit einem $b$ überschreibt.

$M$ ist deterministisch, weil $\delta$ eine totale Funktion ist. Die Berechnung läuft bis $M$ in $h$ ist ($M$ *hält*), oder $M$ sich links von dem am weitesten links stehenden Zeichen bewegt ($M$ *hängt*).

**Beispiel**

Sei $M$ die Maschine $\langle K, \Sigma, \delta, q\rangle$ mit

$K = \{q\}$,

$\Sigma = \{a, \#\}$,

$\delta(q, a) = \langle q, L\rangle$ und

$\delta(q, \#) = \langle h, \#\rangle$.

Diese Maschine bewegt sich nach links bis sie ein # sieht. Wenn links von der (vorläufig beliebigen) Startkonfiguration kein # ist, dann hängt die Maschine.

**Konfigurationen**

Eine Konfiguration einer Turingmaschine $M$ ist ein 4-Tupel

$$\langle p, u, \sigma, v\rangle$$

und besteht aus:

$p$ – dem aktuellen Zustand

$u$ – dem Teil des Bandes *links* von der aktuellen Position

$\sigma$ – dem in der aktuellen Position gelesenen Zeichen

$v$ – dem Teil des Bandes *rechts* von der aktuellen Position, aber nur bis einschließlich dem letzten nicht-blanken Zeichen.

Wenn z.B. eine Turingmaschine $M$ in einem Zustand $p$ ist, das Eingabeband *##aabaa##…* ist, und $M$ sich in der Eingabe am $b$ befindet, dann ist die Konfiguration $\langle p, \#\#aa, b, aa\rangle$. Wenn $p = h$, dann ist die Konfiguration eine *Haltekonfiguration*.

**Vereinfachte Notation für Konfigurationen**

Statt $\langle p, u, \sigma, v\rangle$ schreiben wir $\langle p, u\underline{\sigma}v\rangle$; z.B. schreiben wir $\langle p, \#\#aa, b, aa\rangle$ einfacher als $\langle p, \#\#aa\underline{b}aa\rangle$.

**Die ergibt-Relation für Turing-Maschinen**

Wir sparen uns die volle Formalität der Definition, und formulieren es lieber so:

Sei $M = \langle K, \Sigma, \delta, q\rangle$ eine Turing-Maschine, sei $\langle q, u\underline{a}v\rangle$ eine Konfiguration von $M$ und sei $\delta(q, a) = \langle p, b\rangle$. Dann gilt:

Wenn $b \in \Sigma$:
$\langle q, u\underline{a}v\rangle \vdash_M \langle p, u\underline{b}v\rangle$

Wenn $b = L$:
Wenn $u = u'c$ für ein $u' \in \Sigma^*$ und $c \in \Sigma$:
Wenn $av \neq \#$: $\langle q, u'c\underline{a}v\rangle \vdash_M \langle p, u'\underline{c}av\rangle$

Wenn $av = \#$: $\langle q, u'c\underline{\#}\rangle \vdash_M \langle p, u'\underline{c}\rangle$

Sonst ($u = \varepsilon$) hängt die Maschine und wir nennen $\langle q, u\underline{a}v\rangle$ *eine Hängekonfiguration.*

Wenn $b = R$:
Wenn $v = cv'$ für ein $c \in \Sigma$ und $v' \in \Sigma^*$:
$\langle q, u\underline{a}v\rangle \vdash_M \langle p, uac\underline{v}'\rangle$
Sonst ($v = \varepsilon$):
$\langle q, u\underline{a}\rangle \vdash_M \langle p, ua\underline{\#}\rangle$

Die ergibt-in-einem-Schritt-Relation ist durch dies vollständig beschrieben.

$\vdash_M^*$ ist wie immer die reflexiv-transitive Hülle von $\vdash_M$.

### Eingabe für Turing-Maschinen

Wir werden jetzt die Eingabe einer Turing-Maschine standardisieren: Sei $M$ eine Turingmaschine, und $w$ eine Zeichenkette über dem Alphabet der Maschine:

### Definition

Eine Turingmaschine $M$ *hält bei Eingabe* $w$

gdw

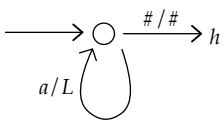
$\langle s, \#w\underline{\#}\rangle$ ergibt eine Haltekonfiguration.

$M$ *hängt bei Eingabe* $w$ gdw $\langle s, \#w\underline{\#}\rangle$ eine Hängekonfiguration ergibt.

### Zustandsdiagramme

Wir können DEA-ähnliche Zustandsdiagramme für Turing-Maschinen einführen. Nehmen wir an, $\delta(q, a) = \langle p, b\rangle$. Der Pfeil zwischen $q$ und $p$ wird dann mit $a/b$ markiert.

Hier z.B. ein Zustandsdiagramm für die Maschine aus dem ersten Beispiel:



## Berechnungen mit Turing-Maschinen

### Turing-Berechenbarkeit

Seien $\Sigma_1$ und $\Sigma_2$ Alphabete ohne # (# $\notin \Sigma_1$, # $\notin \Sigma_2$) und sei $f$ eine Funktion von $\Sigma_1^*$ in $\Sigma_2^*$.

Eine Turing-Maschine $M = \langle K, \Sigma, \delta, s\rangle$ *berechnet* $f$

gdw

Für jedes $w$ in $\Sigma_1^*$: $\langle s, \#w\underline{\#}\rangle \vdash_M^* \langle h, \#f(w)\underline{\#}\rangle$

Wenn es zu $f$ eine Turingmaschine $M$ gibt, die $f$ berechnet, heißt $f$ *Turing-berechenbar.*

Erweiterung auf mehrstellige Funktionen:

Eine mehrstellige Funktion $f: (\Sigma_1^*)^n \rightarrow \Sigma_2^*$ ($n \geq 0$) kann ähnlich berechnet werden; eine solche Funktion wird von $M$ berechnet gdw

Für jedes $w$ in $(\Sigma_1^*)^n$:
$\langle s, \#w_1\#w_2 \ldots \#w_n\underline{\#}\rangle \vdash_M^* \langle h, \#f(w_1, w_2, \ldots, w_n)\underline{\#}\rangle$

Zahlenfunktionen:

Zahlenfunktionen lassen sich als ein Sonderfall der Zeichenkettenfunktionen behandeln. Wir brauchen nur ein Zahlensystem festzulegen, z.B. können wir unäre Repräsentation benutzen (0 als ε, 1 als 1, 2 als 11, 3 als 111 usw.).

### Turing-akzeptierbar

Sei $\Sigma$ ein Alphabet ohne #. *M akzeptiert* $w \in \Sigma^*$ gdw $M$ hält bei Eingabe $w$.

Sei $L$ eine Sprache über einem Alphabet $\Sigma$. *M akzeptiert* $L$ gdw $L = \{w \in \Sigma^*$*: M akzeptiert* $w\}$.

*L ist Turing-akzeptabel* gdw es eine Turing-Maschine gibt, die $L$ akzeptiert.

### Turing-entscheidbar

Eine Sprache $L$ über $\Sigma$ ist *Turing-entscheidbar,* wenn es eine Turingmaschine gibt, die für jede Zeichenkette $w$ über $\Sigma$ entscheiden kann, ob $w \in L$ oder $w \notin L$. D.h., die Turingmaschine kann die charakteristische Funktion $\chi_L$ der Menge $L$ berechnen. Wir definieren Turing-Entscheidbarkeit deshalb via $\chi_L$: Sei $\Sigma$ ein Alphabet ohne #, und seien Ⓨ und Ⓝ Symbole, die in $\Sigma$ nicht vorkommen. Definiere $\chi_L$ von $\Sigma^*$ in {Ⓨ, Ⓝ} folgendermaßen:

$\chi_L(w) =$ Ⓨ, *wenn* w ∈ L

$\chi_L(w) =$ Ⓝ, *wenn* w ∉ L

$L$ ist *Turing-entscheidbar* gdw $\chi_L$ Turing-berechenbar ist.

$M$ *entscheidet* $L$ gdw $M$ $\chi_L$ berechnet.

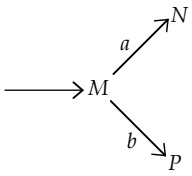
### Zusammengesetzte Turingmaschinen

Wenn $M$ und $N$ zwei Turingmaschinen sind, ist $M \to N$ (oder einfach $MN$) die Turingmaschine, die erst die Berechnung von $M$ durchführt, und falls $M$ halten würde, in den Startzustand von $N$ übergeht. Wenn $M$ und $N$ Turingmaschinen sind, ist

$$M \xrightarrow{a} N$$

die Maschine, die erst die Berechnung von $M$ durchführt. Falls $M$ mit einem $a$ unter dem Lesekopf halten würde, geht $M \xrightarrow{a} N$ in den Startzustand von $N$ über, sonst hält sie.

Entsprechend können wir auch Maschinen bilden, die je nach gelesenem Zeichen in einer von mehreren Maschinen ihre Berechnung fortsetzt, wie:



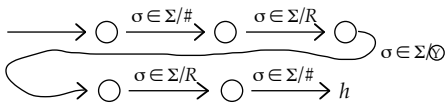

### Basismaschinen

Für die drei möglichen Transitionstypen führen wir Basismaschinen ein:

- $L$ ist für jedes Alphabet $\Sigma$ die Maschine, die für jedes Zeichen $a$ in $\Sigma$ die Transition $\delta(s, a) = \langle h, L\rangle$ hat (und keine anderen Transitionen hat).
- $R$ ist für jedes Alphabet $\Sigma$ die Maschine, die für jedes Zeichen $a$ in $\Sigma$ die Transition $\delta(s, a) = \langle h, R\rangle$ hat (und keine anderen Transitionen hat).
- $\sigma$ ist für jedes Alphabet $\Sigma$ die Maschine, die für jedes Zeichen $a$ in $\Sigma$ die Transition $\delta(s, a) = \langle h, \sigma\rangle$ hat (und keine anderen Transitionen hat).

### Beispiel

Die Maschine $\#R\text{Ⓨ}R\#$ kommt als Teilmaschine in vielen Turingmaschinen vor. Wir hätten auch die Maschine folgendermaßen darstellen können:



Die Notation #*R*Ⓨ*R*# ist etwas gewöhnungsbedürftig, aber sehr platzsparend!

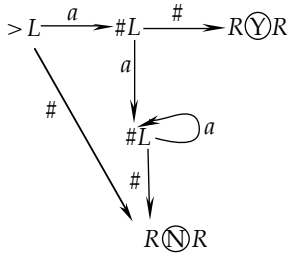
### Einige nützliche Teilmaschinen

Eine nützliche Erweiterung der Notation ist folgende: Wenn über einem Pfeil ein Zeichen $\overline{a}$ vorkommt, bedeutet dies, dass dem Pfeil bei jedem anderen Zeichen als a gefolgt werden soll.

Die Maschine $L_\sigma$ ist für jedes $\sigma$ die Maschine, die sich nach links bewegt, bis sie ein $\sigma$ findet. Besonders viel Anwendung findet $L_\#$, die folgendermaßen aussieht:


### Beispiel

Diese Maschine entscheidet die Sprache {*a*} (und arbeitet mit dem Alphabet {*a*, #, Ⓨ , Ⓝ}):



### Nicht-deterministische Turing-Maschinen

sind identisch mit den normalen Turing-Maschinen, außer dass $\delta$ nicht mehr eine Funktion sein muss, sondern eine Relation sein kann (und wird dann normalerweise mit $\Delta$ bezeichnet).

Zu jeder Sprache $L$ die von einer nicht-deterministischen Turing-Maschine akzeptiert wird, gibt es eine deterministische Turing-Maschine die $L$ akzeptiert (wird nicht bewiesen).

### Universalmaschinen

Problem: Eine Turing-Maschine $U$ zu finden, die Berechnungen jeder beliebigen Turing-Maschine simulieren kann.

Zunächst müssen wir die Annahme machen, dass es (abzählbar unendliche) Mengen $V_K = \{q_0, q_1, \ldots\}$ von Zuständen (wobei $q_0 = h$) und $V_\Sigma = \{\sigma_0, \sigma_1, \ldots\}$ von Zeichen (wobei $\sigma_0 = \#$) gibt, so dass jede Turing-Maschine $M$ ein Alphabet $\Sigma \subseteq V_\Sigma$ und eine Zustandsmenge $K \subseteq V_K$ hat. Diese Annahme ist harmlos: Jede Turingmaschine hat ein endliches Alphabet $\Sigma$ und endlich Menge von Zuständen $K$, die immer durch endliche Teilmengen von $V_\Sigma$ bzw. $V_K$ repräsentiert werden können.

Die Idee ist jetzt, wie in richtigen Computern die Zeichen und Zustände als Bytes, d.h. als eine Zeichenkette über $\{0, 1\}$ zu repräsentieren. (Hier weichen wir von L&P etwas ab.)

Wir müssen dann erst die für eine Maschine $M$ erforderliche Bytelänge finden. Sei $i$ die größte Zahl, so dass $\sigma_i \in \Sigma$ oder $q_i \in K$. Die Bytelänge ist dann die kleinste Zahl $l$ so dass $2^l > i$.

Wir kodieren jetzt $q_j$ und $\sigma_j$ beide als $j$ in binärer Darstellung, rechtsgerückt und links mit Nullen auf Länge $l$ verlängert. Diese binäre Darstellung von $j$ bezeichen wir mit $\beta(j, l)$.

### Kodieren vom Band und Lesekopf

Den nicht-blanken Teil des Bandes, $w = \sigma_{i1}\ldots\sigma_{in}$, werden wir als

$$\rho(w) = B\beta(i_1, l)B\beta(i_2, l)B\ldots B\beta(i_n, l)B$$

kodieren.

Die Position des Lesekopfes wird folgendermaßen kodiert: Rechts der Repräsentation des Zeichens das $M$ gerade sieht, steht statt $B$ ein $b$.

### Kodieren von Transitionen

Eine Transition $\delta(q_i, \sigma_j) = \langle q_n, \sigma_m \rangle$ wird als

$D\beta(i, l)D\beta(j, l)D\beta(n, l)D\beta(m, l)D$

kodiert. Eine Transition $\delta(q_i, \sigma_j) = \langle q_n, R\rangle$ wird als

$D\beta(i, l)D\beta(j, l)D\beta(n, l)DRD$

kodiert, entsprechend auch für $L$.

### Kodieren von δ

Die ganze Funktion $\delta$, mit $n = |K||\Sigma|$ Transitionen, wird als

$\rho(\delta) = T_1 \ldots T_n$

kodiert, wobei $T_1, \ldots, T_n$ die Kodierungen der Transitionen sind. Damit sind die Kodierungen der Transitionen durch zwei $D$'s getrennt.

### Kodieren vom Startzustand und Maschine:

Der Startzustand $s=q_j$ wird einfach durch

$\rho(s) = Z\beta(j, l)Z$

kodiert. Mehr brauchen wir nicht, um eine Maschine vollständig zu repräsentieren. Was die Mengen $K$ und $\Sigma$ sind, ergibt sich implizit. Also setzen wir

$\rho(M) = \rho(\delta)\rho(s)$.

### Die vollständige Eingabe für die Universalmaschine

Um eine Berechnung von $M$ bei Eingabe $w$ zu simulieren, wird $U$ in der Konfiguration

$\langle s, \#\rho(M)\rho(w)\#\rangle$

gestartet. Das Feld mit dem Startzustand wird später benutzt, um dort den aktuellen Zustand von $M$ zu speichern. Wenn dort $\beta(0, l)$ erscheint, ist $M$ im Haltezustand, und $U$ hält auch, und zwar in der Konfiguration

$\langle h, \#\rho(\delta)Z\beta(0, l)Z\rho(w')\#\rangle$

wobei $w'$ das Band von $M$ beim Halten ist.

### Beispiel

Nehmen wir als Beispiel eine Maschine, die nach rechts nach dem ersten Vorkommen von zwei $a$'s sucht, und nehmen wir an, dass $\sigma_1 = a$ und $\sigma_2 = b$. Die Maschine ist $M = \langle\{q_1, q_2\}, \{a, b, \#\}, \delta, q_1\rangle$ wobei

$\delta(q_1, a) = \langle q_2, R\rangle$
$\delta(q_1, b) = \langle q_1, R\rangle$
$\delta(q_1, \#) = \langle q_1, R\rangle$
$\delta(q_2, a) = \langle h, a\rangle$
$\delta(q_2, b) = \langle q_1, R\rangle$
$\delta(q_2, \#) = \langle q_1, R\rangle$

Die erforderliche Bytelänge ist dann 2, und wir erhalten:

*ρ(δ) = D01D01D10DRDD01D10D01DRD-D01D00D01DRD D10D01D00D01D-D10D10D01DRDD10D00D01DRD*

Wenn die Startkonfiguration von $M$ $\langle q_1, aab \rangle$ ist, wird $U$ in folgender Konfiguration gestartet:

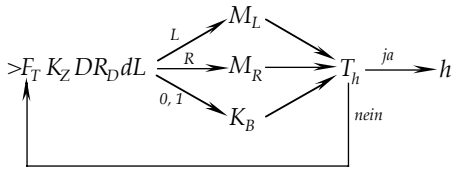
*⟨s, #D01D01D10DRDD01D10D01DRD-D01D00D01DRD D10D01D00D01D-D10D10D01DRDD10D00D01DRD Z01Z-B01b01B10B#⟩*

### Zusätzliche Hilfszeichen

Bis jetzt haben wir die Zeichen *#, 0, 1, L, R, D, Z, B, b* benutzt. Als zusätzliche Hilfszeichen führen wir *d, o* und *i* ein. *d* wird als Markierung beim Auffinden von der richtigen Transition benutzt. *o* und *i* werden bei Kopierverfahren als "markierte" Ausgaben von *0* und *1* benutzt.

### Die Universalmaschine U

sieht in den Grundzügen folgendermaßen aus:




Die Teilmaschine $F_T$ ("finde Transition") findet die Transition, die mit der Kopfposition und dem Zustand übereinstimmt. Die Transition wird markiert, in dem das *D* nach der Kodierung des neuen Zustandes mit einem *d* ersetzt wird.

Beispiel: Wenn das Band vor der Berechnung von $F_T$ so aussieht:

*#D101D001D100DLDD…DZ101ZB011B001b#*

sieht es nach der Berechnung folgendermaßen aus:

*#D101D001D100dLDD…DZ101ZB011B001b*

Die Teilmaschine $K_Z$ ("kopiere Zustand") kopiert den Zustand links vom *d* zum "Zustandsspeicher" zwischen den beiden *Z*'s, und hält auf dem *d*.

Beispiel: Das Band sieht jetzt so aus:

*#D101D001D100dLDD…DZ100ZB011B001b*

Nach diesen Teilmaschinen kommt die Sequenz *DRDdL*. Hier wird das *d* in ein *D* umgewandelt. Dann wird das nächste *D* rechts gesucht und in ein

*d* umgewandelt. Die Maschine steht jetzt direkt rechts von der Kodierung der Aktion, die *M* ausführen soll (Zeichen schreiben, links oder rechts gehen). Es wird getestet, ob links ein *L*, ein *R* oder die Kodierung eines Zeichens steht. Dann wird entweder eine Bewegungs-Teilmaschine ($M_R$ oder $M_L$) oder eine Zeichenkopierungs-Teilmaschine ($K_B$, "kopiere zum Band") benutzt.

Jetzt sieht unser Beispielband folgendermaßen aus:

*#D101D001D100DLDD…DZ100ZB011b001B#*

Schließlich wird getestet ob der aktuelle Zustand der Haltezustand ist (steht zwischen *Z* und *Z* nur Nullen?), wenn nein, wird eine neue Schleife angefangen, wenn ja, hält die Maschine.

### Das Halteproblem für Turing-Maschinen

Das Halteproblem ist das folgende: Gibt es eine Turing-Maschine, die für jede Turing-Maschine *M* und jede Zeichenkette *w* entscheiden kann, ob M bei Eingabe *w* hält, das heißt, ob *M w* akzeptiert?

Das Halteproblem ist nicht lösbar. Um dies zu Zeigen führen wir die Sprache $K_0$ ein:

$K_0 = \{\rho(M)\rho(w)$: *M akzeptiert w*$\}$

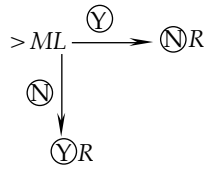
Aber erst zwei Lemmas:

### Lemma 1

Wenn *L* Turing-entscheidbar ist, ist $\Sigma^*$ – *L* es auch.

### Beweis

Wenn *L* von *M* entschieden wird, wird $\Sigma^*$ – *L* von




entschieden. □

### Lemma 2

Wenn *L* Turing-entscheidbar ist, ist *L* Turingakzeptabel.

### Beweis

Wenn *M L* entscheidet, wird *L* von dieser Maschine akzeptiert:


**Theorem**

Keine Turingmaschine kann die Sprache $K_0$ entscheiden.

**Beweis**

Nehmen wir an, $K_0$ ist Turing-entscheidbar. Dann ist auch

$K_1 = \{\rho(M): M \text{ akzeptiert } \rho(M)\}$

Turing-entscheidbar: Wenn $K_0$ von $M_0$ entschieden wird, können wir eine Maschine $M'$ konstruieren, die aus $w$ erst $w\rho(w)$ berechnet, und dann dies als Eingabe für $M_0$ weitergibt.

Dann ist aber auch (nach Lemma 1) $\overline{K_1} = \Sigma^* - K_1$ Turing-entscheidbar, nach Lemma 2 ist $\overline{K_1}$ dann auch Turing-akzeptabel. Wir zeigen, dass dies nicht der Fall sein kann (!):

Sei $M^*$ eine Turing-Maschine die $\overline{K_1}$ akzeptiert. Dann gilt:

$\rho(M^*) \in \overline{K_1}$

*gdw $M^*$ akzeptiert nicht $\rho(M^*)$ (Def. $\overline{K_1}$)*

gdw $\rho(M^*) \notin \overline{K_1}$
($\overline{K_1}$ ist die akzeptierte Sprache von M*)

Kontradiktion! Dies heißt, dass $\overline{K_1}$ nicht Turing-akzeptabel sein kann. Deshalb ist $\overline{K_1}$ nicht Turing-*entscheidbar*, und dann ist auch $\overline{K_0}$ nicht Turing-entscheidbar. □

**Korrolar**

Es gibt Sprachen (z.B. $\overline{K_1}$), die nicht Turingakzeptierbar sind. □

**Theorem**

Es gibt Turing-akzeptierbare Sprachen, die nicht Turing-entscheidbar sind.

**Beweis**

Wir zeigen, dass $K_0$ Turing-akzeptabel ist:

Die Universalmaschine $U$ ist fast eine Maschine, die $K_0$ akzeptiert.

Wenn $U$ bei Eingabe $w$ hält, ist es entweder, weil die simulierte Maschine hält (dann ist $w$ eine Kodierung einer Turingmaschine mit Eingabe und $w \in K_0$) oder

(möglicherweise) weil die Eingabe nicht die Kodierung einer Turing-Maschine mit Eingabe ist.

Den letzten Fall müssen wir verhindern. Also bauen wir eine Maschine $S_U$, die nur Eingaben akzeptiert, die korrekte Kodierungen sind (wir lassen die Konstruktion von $S_U$ hier aus).

$K_0$ wird dann von der zusammengesetzten Maschine $S_U U$ akzeptiert. □

#### Korrolar

Die Turing-akzeptierbaren Sprachen sind nicht unter Komplementbildung abgeschlossen.

#### Beweis

$K_1$ ist Turing-akzeptabel, weil $K_0$ es ist. $\overline{K_1}$ ist aber nicht Turing-akzeptabel. □

## Die rekursiv aufzählbaren Sprachen und die rekursiven Sprachen

Wenn die Church'sche These wahr ist, bedeutet dies, dass die Begriffe *berechenbar* und *Turing-berechenbar* identisch sind. Weil bis jetzt nichts gegen die Church'sche These spricht, werden deshalb viele Begriffe von Berechenbarkeit via Turingmaschinen definiert:

#### Definition

Eine Sprache heißt *rekursiv aufzählbar*,[3] wenn sie von einer Turing-Maschine akzeptiert wird.

Wir können jetzt auch unsere Definition der *rekursiven* Sprachen präziser machen:

#### Definition (Präzisierung)

Eine Sprache heißt rekursiv, wenn sie von einer Turing-Maschine entschieden wird.

#### Typ-0-Grammatiken und Turing-Maschinen

*Theorem*

Die Klasse der rekursiv aufzählbaren Sprachen und die Klasse der Typ-0-Sprachen (von Typ-0-Grammatiken generierten Sprachen) sind identisch.

*Beweis (Skizze)*

Erst die eine Richtung:

Zu jeder Typ-0-Grammatik $G$ gibt es eine Turing-Maschine die $L(G)$ akzeptiert.

3 Eine Sprache wird von einer Turing-Maschine aufgezählt gdw sie von einer Turing- Maschine akzeptiert wird (wird nicht gezeigt).

Sei $G$ eine Typ-0-Grammatik. Wir können eine Turing-Maschine $M_G$ folgendermaßen konstruieren:

$M_G$ behält während der ganzen Berechnung das Eingabewort $w$ auf dem Band.

$M_G$ erzeugt jede mögliche Zeichenkette über $(V \cup \{\Rightarrow\})^*$, geordnet nach Länge, und für jede Länge in lexikographischer Reihenfolge (z.B. $a$, $S$, $\Rightarrow$, $aa$, $aS$, $a\Rightarrow$, $Sa$, $SS$, $S\Rightarrow$, $\Rightarrow a$, $\Rightarrow S$, $\Rightarrow\Rightarrow$, $aaa$, … usw.).

Für jede erzeugte Zeichenkette überprüft $M_G$, ob die Zeichenkette eine $G$-Ableitung ist, und ob $w$ das letzte Wort der Ableitung ist. Wenn ja, hält $M_G$, wenn nein setzt sie die Generierung fort.

Dann die andere Richtung:

Zu jeder Turing-Maschine gibt es eine Grammatik, die die von $M$ akzeptierte Sprache generiert.

Die Idee ist, eine Konfiguration $\langle q, u, a, v\rangle$ mit der Zeichenkette $[uqav]$ zu repräsentieren. Wir nehmen an, dass die Zustandssymbole nicht Alphabetsymbole sind. Also ist in der Zeichenkette das Zustandssymbol als solches erkennbar, und es markiert gleichzeitig die Position des Lesekopfes (rechts von dem Zustand).

Erst führen wir Regeln ein, mit denen wir jede Zeichenkette

$w[\#ws\#]$

erzeugen können, wo $w$ eine beliebige Zeichenkette über dem Alphabet der Turing-Maschine ist. Diesen Teil der Grammatik können wir sehr ähnlich einer Grammatik für *COPY* machen (wie in der Übung!), wir müssen nur aufpassen, dass die Regeln nicht auch später angewandt werden können, dass die linke Kopie von $w$ in $w[\#ws\#]$ bei weiteren Ableitungen unberührt bleibt.

Dann führen wir für jede Transition $\delta(q, a) = \langle p, b\rangle$ in $M$ die Regel $qa \to pb$ ein (schreibt z.B. $aa[\#aqa]$ in $aa[\#apb]$ um), und für jede Transition $\delta(q, a) = \langle p, R\rangle$ die Regeln $qab \to apb$ (für jedes $b$ im Alphabet der Turing-Maschine) und $qa] \to ap\#]$, ähnliche Regeln werden auch für $L$-Bewegungen eingeführt.

Schließlich führen wir Regeln ein, die, falls ein $h$ auftaucht, den Teil zwischen [ und ] löscht, z.B.

$ha \to h$ und $ah \to h$

für beliebige Zeichen $a$ außer [ und ], und schließlich

$[h] \to \varepsilon$.

Wenn nun die Grammatik genau das Alphabet der Turing-Maschine als Nichtterminale hat, sind die

einzigen Zeichenketten aus Terminalzeichen genau diejenigen, wo zuletzt die Regel $[h] \to \varepsilon$ angewandt wurde, also diejenigen Zeichenketten, die die Turing-Maschine akzeptiert. □

### Schlussbemerkung

Wir haben die folgenden Sprachklassen mit zugehörenden Automaten und Grammatiken studiert: Die Sprachklassen sind (mit der Ausnahme von kontextfreien Sprachen mit ε) von oben nach unten durch echte Inklusion geordnet. Ein „echtes Beispiel“ ist ein Beispiel einer Sprache der aktuellen Klasse, die nicht zu der engeren Klasse gehört:

| *Sprachklasse* | *"echtes" Beispiel* | *Grammatik-Typ* | *Charakterisierung durch Automaten* |
|---|---|---|---|
| Reguläre Sprachen | $a^*$ | Typ 3 | D(N)EA akzeptiert |
| Det. kontextfreie Spr. | $\{a^n b^n : n \geq 0\}$ | — | DKA akzeptiert |
| Kontextfreie Spr. | *Kompl. von* $\{a^n b^n c^n : n \geq 0\}$ | Typ 2 | KA akzeptiert |
| Kontextsensitive Spr. | $\{a^n b^n c^n : n \geq 0\}$ | Typ 1 | *LBA (nicht vorgestellt)* |
| Rekursive Sprachen | *(nicht vorgestellt)* | — | Turing-M. entscheidet |
| Rekurs. aufzählb. Spr. | $K_0$ | Typ 0 | Turing-M. akzeptiert |
| Nicht r.k.a.S. | *Kompl. von* $K_1$ | — | Keine TM akz. |