5.1 Formale Sprachen und Automaten

5.1.1 Formale, natürliche und Programmiersprachen

Die **Theorie der formalen Sprachen** und die **Automatentheorie** nehmen in der theoretischen Informatik breiten Raum ein. Dabei werden die Einsichten vermittelt, die zum Verständnis von Anwendungen in verschiedenen anderen Gebieten erforderlich sind. Dies gilt beispielsweise für die Linguistik, den Compilerbau und die künstliche Intelligenz.

Linguistik:
Sprachwissenschaft

> Die Wortwahl **formale Sprache** unterstreicht die Abgrenzung sowohl zu den natürlichen als auch zu den Programmiersprachen.
> In formalen Sprachen spielt die **Semantik,** d.h. die Bedeutung der Sätze einer Sprache, *keine* Rolle.

In der Programmiersprache Turbo Pascal bedeutet

```
for i:= 1 to 5 do writeln('Hallo');
```

dass das Wort Hallo am Bildschirm 5-mal untereinander geschrieben wird.
Das for-to-do-Sprachelement hat die Bedeutung eines Zyklus. Die darin enthaltene Wertzuweisung folgt der Semantik von `:=`.

Für die Definition der Semantik von Programmiersprachen sind verschiedene Zugänge entwickelt worden, die hier jedoch nicht betrachtet werden.

5.1.2 Syntax und Ableitungsbaum

Die Theorie der formalen Sprachen befasst sich ausschließlich mit dem syntaktischen Aspekt.

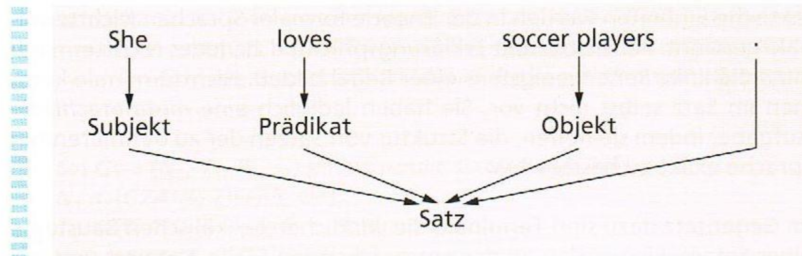
> Die **Syntax** einer Sprache ist die Lehre vom Satzbau. Sie umfasst sämtliche grammatikalische Regeln, deren Anwendungen zu korrekt gebauten Sätzen führen.

Auch natürliche und Programmiersprachen besitzen eine Syntax.

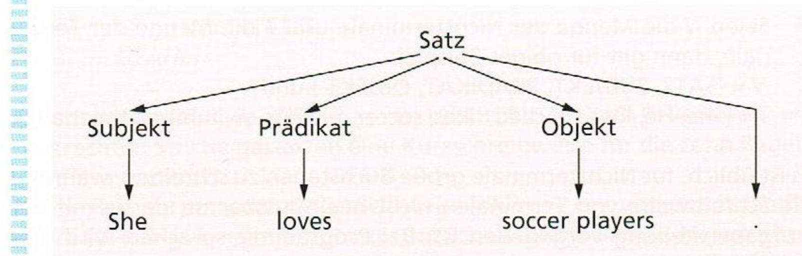
Für englische Sätze gilt die SPO-Regel, die die Reihenfolge Subjekt – Prädikat – Objekt vorschreibt.

Das Beispiel führt zu der Frage, wie grammatikalische oder syntaktische Regeln anzuwenden sind. Dafür gibt es zwei Herangehensweisen:
1. Erzeugung (Synthese) eines Satzes
2. Erkennung (Analyse) eines Satzes

Der Satz „She loves soccer players." ist syntaktisch korrekt, denn die Analyse gemäß der SPO-Regel verläuft positiv:



Ein „Konstrukteur" dieses Satzes geht umgekehrt vor:



Die für die Analyse bzw. Synthese eines Satzes verwendete Darstellung heißt **Baum**. Sogenannte **Ableitungsbäume** entstehen aber nur dann, wenn sämtliche Regeln eine spezielle Form besitzen (↗ Abschnitte 5.1.3 und 5.1.7: kontextfreie Grammatik).

Die Satzbauregel für englische Sätze lautet in Kurzform
SATZ → SUBJEKT PRÄDIKAT OBJEKT.
Lies: „Ein SATZ *ist definiert als* eine Folge aus SUBJEKT, PRÄDIKAT, OBJEKT und einem anschließenden Punkt."
Diese eine Regel genügt noch nicht, denn sie gibt keine Auskunft darüber, wofür SUBJEKT, PRÄDIKAT und OBJEKT stehen. Dies legen die folgenden Regeln fest:

SUBJEKT → She
SUBJEKT → He
PRÄDIKAT → loves
PRÄDIKAT → kicked
PRÄDIKAT → likes
OBJEKT → soccer players
OBJEKT → swimming
OBJEKT → the ball

Damit können insgesamt 18 Sätze gebildet werden, wobei ein Satz wie „She kicked soccer players." daran erinnert, dass die Semantik wirklich keine Rolle spielt.

Für das Zeichen → („ist definiert als") wird oft auch das Zeichen ::= benutzt.

5.1.3 Formale Grammatik

Offensichtlich gelingt es immer dann (wenigstens) einen Ableitungsbaum für einen gegebenen Satz anzugeben, wenn in jeder Regel auf der linken Seite genau eine zu erklärende **syntaktische Einheit** steht. Syn-

taktische Einheiten werden in der Theorie formaler Sprachen **Nichtterminale** genannt. Für sie besteht Erklärungspflicht, d. h., jedes Nichtterminal muss die linke Seite wenigstens einer Regel bilden. Nichtterminale kommen im Satz selbst nicht vor. Sie haben lediglich eine *metasprachliche* Aufgabe, indem sie helfen, die Struktur von Sätzen der zu definierenden Sprache exakt zu beschreiben.

Im Gegensatz dazu sind **Terminale** die wirklichen lexikalischen Bausteine eines Satzes. Sie werden an der entsprechenden Stelle platziert und mit anderen Terminalen verbunden.

Seien *N* die Menge der Nichtterminale und *T* die Menge der Terminale, dann gilt für obiges Beispiel:
N = {SATZ, SUBJEKT, PRÄDIKAT, OBJEKT } und
T = {She, He, loves, kicked, likes, soccer_players, swimming, the_ball,.}.

Es ist üblich, für Nichtterminale große Buchstaben zu schreiben, während die Schreibweise von Terminalen nicht beeinflussbar ist, da sie mit der Aufgabenstellung vorgegeben ist. Bei Programmiersprachen wird dies vom „Design der Sprache" vorgeschrieben. In den folgenden Beispielen werden meist Kleinbuchstaben als Terminalsymbole benutzt.
Im letzten Beispiel spielt das Nichtterminal SATZ eine besondere Rolle. Es bildet in jedem Falle die **Wurzel des Ableitungsbaumes** und heißt deshalb **Axiom, Satz-, Spitzen-** oder **Startsymbol** s.
Eine Satzerzeugung beginnt also stets beim Spitzensymbol und endet, wenn sämtliche Nichtterminale durch Terminale ersetzt worden sind. Anstelle der Baumform gibt es noch folgende Schreibweise:
$s \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$, kurz $s \overset{*}{\Rightarrow} \alpha_n$.
Die α_i werden **Satzformen** genannt. Es sind Zeichenketten, die aus beliebig vielen Nichtterminalen und beliebig vielen Terminalen – auch gemischt und in beliebiger Reihenfolge – bestehen.
Da die **Ableitung eines Satzes** – manchmal auch Herleitung genannt – bei α_n endet, ist klar, dass α_n kein einziges Nichtterminal enthält. α_n ist ein *aus s abgeleiteter* Satz.

Achtung!
Der Ableitungspfeil in $\alpha_i \Rightarrow \alpha_j$, sprich: α_i „geht unmittelbar über in" α_j, darf nicht verwechselt werden mit dem Definitionssymbol in den grammatikalischen Regeln, $X \rightarrow \alpha$, sprich X „ist definiert als" α.

Der Satz „He likes swimming." ist aus s=SATZ ableitbar, denn
SATZ ⇒ SUBJEKT PRÄDIKAT OBJEKT . ⇒ He PRÄDIKAT OBJEKT .
⇒ He likes OBJEKT . ⇒ He likes swimming .

Die vorangegangenen Betrachtungen münden nun in die Definitionen einiger abstrakter Begriffe.

Eine **formale Grammatik** $G = (N, T, P, s)$ ist ein 4-Tupel, wobei die verwendeten Symbole die folgenden Bedeutungen besitzen:
- *N* eine endliche Menge von *N*ichtterminalen;
- *T* eine endliche Menge von *T*erminalen, mit $N \cap T = \emptyset$;
- *P* eine endliche Menge von Regeln oder *P*roduktionen der Form $\alpha \rightarrow \beta$, wobei α und β Satzformen sind und α wenigstens aus einem Nichtterminal besteht; und schließlich
- *s* das Startsymbol, mit $s \in N$.

↗ auch Definition einer **formalen Grammatik** im Abschnitt 3.1.4

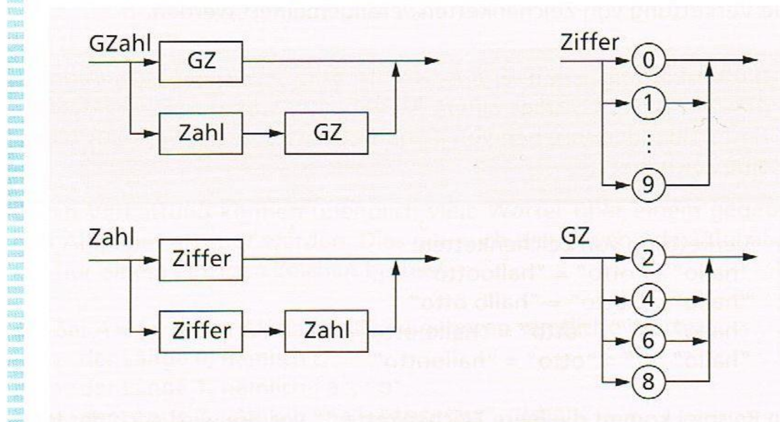
Der amerikanische Informatiker NOAM CHOMSKY (geb. 1928) hat formale Grammatiken erstmals ab 1956 verwendet. Um diese Leistung zu würdigen, werden formale Grammatiken (dieser Art) auch **Chomsky-Grammatiken** genannt.

- Sei $G_1 = (N_1, T_1, P_1, s_1)$ eine formale Grammatik, mit
  $N_1 = \{GZAHL, ZIFFER, GZ\}$,
  $T_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$,
  $P_1 = \{GZAHL \rightarrow GZ \mid ZAHL\ GZ$,
  $ZAHL \rightarrow ZIFFER \mid ZIFFER\ ZAHL$,
  $ZIFFER \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$,
  $GZ \rightarrow 0 \mid 2 \mid 4 \mid 6 \mid 8\}$,
  $s_1 = GZAHL$.

Mit dieser formalen Grammatik können alle geraden Zahlen (als Wörter) erzeugt bzw. erkannt werden.

Die beispielsweise in $ZIFFER \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$ verwendeten senkrechten Striche gestatten eine Kurzschreibweise für die zehn Regeln $ZIFFER \rightarrow 0$, $ZIFFER \rightarrow 1$, … , $ZIFFER \rightarrow 9$.
Die im Beispiel eingeführte Schreibweise inklusive der Verkürzungen für *alternative Regeln* geht auf BACKUS zurück und wird **Backus-Normalform** oder **Backus-Naur-Form**, kurz **BNF**, genannt. Kommen noch weitere Verkürzungen zum Einsatz, spricht man von einer **erweiterten BNF**, kurz: **EBNF**. Für die Syntaxdefinition von Programmiersprachen werden auch gern **Syntaxgraphen (Syntaxdiagramme)** benutzt.

- Syntaxgraphen für obige Grammatik G_1:



Terminale werden als Kreise bzw. Ovale, **Nichtterminale** werden als Rechtecke dargestellt.

5.1.4 Zeichen, Alphabet, Verkettung, Zeichenkette

Die Operation **Verkettung** legt fest, wie einzelne Zeichen zu einer Zeichenkette verbunden werden.
Die Zeichen stammen aus einem Alphabet.

> Ein **Alphabet** ist eine beliebige endliche nichtleere Menge, deren Elemente **Zeichen** oder **Symbole** genannt werden.

- Die folgenden Mengen
  A_1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, A_2 = {△, ♣, ○},
  A_3 = { sin, cos, tan, cot, int, sqrt }
  sind Alphabete, wobei insbesondere die Elemente in A_3 als (atomare) Symbole, die nicht etwa aus mehreren Buchstabensymbolen zusammengesetzt wurden, anzusehen sind.
  B_1 = {0, 1, 1, 2} und B_2 = { } = Ø
  sind keine Alphabete, da B_1 gegen die Mengendefinition verstößt und B_2 gegen die Alphabetdefinition, wonach die Grundmenge mindestens ein Zeichen enthalten muss.

Das Symbol für den Verkettungsoperator ist ∘, sprich: Kringel. Ähnlich, wie der Punkt für den Multiplikationsoperator in der Mathematik, wird ∘ manchmal weggelassen. Es entstehen Zeichenketten.

- 1 ∘ 3 = "13", 8 ∘ 2 = "82", △ ∘ ♣ = "△♣", sqrt ∘ cot = "sqrtcot"

Die **Verkettung** ∘, oder **Konkatenation** von z_1 und z_2, kurz: $z_1 \circ z_2 = k$, ist eine *zweistellige Operation für Zeichen* aus einem Alphabet A, wobei $k = z_1 z_2$ durch Hintereinanderschreiben von z_1 und z_2 entsteht. k ist eine Zeichenkette, die zum besseren Erkennen in Hochkommata eingeschlossen wird. Selbstverständlich gilt $k \notin A$.

Um ganze Sätze bilden zu können, muss die Verkettung für Zeichen auf die Verkettung von Zeichenketten verallgemeinert werden.

Die **Verkettung** ∘, mit $k_1 \circ k_2 = k$, ist eine *zweistellige Operation für Zeichenketten* über einem Alphabet A, wobei $k = k_1 k_2$ durch Hintereinanderschreiben von k_1 und k_2 entsteht. Das Ergebnis k ist eine Zeichenkette.

- Verkettung von Zeichenketten:
  "hallo" ∘ "otto" = "hallootto"
  "hallo" ∘ " otto" = "hallo otto"
  "hallo" ∘ " " ∘ "otto" = "hallo otto"
  "hallo" ∘ "" ∘ "otto" = "hallootto"

Im Beispiel kommt die **leere Zeichenkette** "" vor. Sie wird auch das **leere Wort** genannt und mit ε bezeichnet. Dies geschieht analog zur leeren Menge { }, für die im Allgemeinen Ø geschrieben wird.

Sei k eine Zeichenkette. Dann gilt:

1. $k \circ \varepsilon = \varepsilon \circ k = k,$
2. $\underbrace{k \circ k \circ k \circ \ldots \circ k}_{\text{n-mal}} = k^n, \quad n \geq 0$
3. $k^0 = \varepsilon$
   $k^n = k \circ k^{n-1} = k^{n-1} \circ k, \quad n > 0$

Die rekursive Definition von k^n unter 3. in obiger Definition enthält die Gleichheit $k \circ k^{n-1} = k^{n-1} \circ k$. Dies gilt natürlich nur für ein und dieselbe Zeichenkette k.

Falsch wäre $k_1 \circ k_2 = k_2 \circ k_1$, denn $\circ$ ist *nicht* kommutativ.

5.1.5 Länge einer Zeichenkette, Wort und Wortmenge

Unter der *Länge einer Zeichenkette k* versteht man die Anzahl aller Zeichen von k. Dabei wird jedes Vorkommen eines bestimmten Alphabetzeichens in der betrachteten Zeichenkette extra gezählt. "abba" und "haus" haben beide die Länge 4.

Die **Länge einer Zeichenkette** $k = z_1z_2z_3 \ldots z_p$, kurz: $|k| = p$, ist gleich der Anzahl der Zeichen, die k enthält. Es ist *nicht* gefordert, dass die vorkommenden Zeichen paarweise verschieden sind. Für $k = \varepsilon$ gilt $|k| = 0$.

Satz: Seien k_1 und k_2, mit $|k_1| = n$ und $|k_2| = m$, Zeichenketten. Dann gilt $|k_1 \circ k_2| = n + m$.

Beweis:

$$\begin{aligned} k_1 &= x_1x_2 \ldots x_n,\ k_2 = y_1y_2 \ldots y_m \\ k_1 \circ k_2 &= x_1x_2 \ldots x_n\, y_1y_2 \ldots y_m \\ &= z_1z_2 \ldots z_nz_{n+1}z_{n+2} \ldots z_{n+m} \\ |k_1k_2| &= n + m \end{aligned}$$

Zeichenketten, die über einem Alphabet A gebildet werden, nennt man **Wörter über A**.

Durch Verkettung können unendlich viele Wörter über einem gegebenen Alphabet erzeugt werden. Dies gilt auch dann, wenn das Alphabet aus nur einem einzigen Zeichen besteht.

Sei $A = \{a, b\}$ ein Alphabet. Dann gehören sämtliche Wörter
- der Länge 0, nämlich ε,
- der Länge 1, nämlich "a", "b",
- der Länge 2, nämlich "aa", "ab", "ba", "bb",
- usw.

in die Menge der Wörter über A, kurz: *Wortmenge über A*.

Offensichtlich können über A r^n verschiedene Wörter der Länge n erzeugt werden, wenn das Alphabet A genau r Elemente besitzt.

Die Menge aller Wörter über einem Alphabet heißt **Wortmenge** A^*, mit $A^* = A^0 \cup A^1 \cup A^2 \cup \ldots \cup A^i \cup \ldots = \sum_{i=0}^{\infty} A^i$

A^* ist stets eine unendliche Menge. Die angegebene Zerlegung von A^* in Teilmengen A^i lässt erahnen, wie man sämtliche Wörter aus A^* längenlexikographisch anordnen kann. Auf diese Weise kann man die Elemente aus A^* durchnummerieren. Mengen, bei denen das gelingt, heißen **abzählbar unendlich**.

Die Definition verwendet A^i für die Menge aller Wörter über A, deren Länge gleich i ist. A^i bedeutet also, dass i-mal je ein beliebiges Element aus A genommen und an die jeweils aktuelle Zeichenkette angehangen wird. Das gleiche Resultat ergibt sich für $i > 0$ durch $A^{i-1} \circ A$. Es macht offenbar Sinn, die Verkettungsoperation auf Mengen anzuwenden.

> Für zwei Mengen M_1 und M_2 gilt
> $M_1 \circ M_2 = \{w_1w_2 \mid w_1 \in M_1 \text{ und } w_2 \in M_2\}$.

- Seien M_1 = {"ab", "cd", "a"} und M_2 = {ε, "x", "xy"}, so ist $M_1 \circ M_2$ = {"ab", "cd", "a", "abx", "cdx", "ax", "abxy", "cdxy", "axy"}.

Die Analogie zum kartesischen Produkt zweier Mengen ist offensichtlich. Ferner gelten $M \circ \emptyset = \emptyset \circ M = \emptyset$ und $M \circ \{\varepsilon\} = \{\varepsilon\} \circ M = M$ für beliebige Mengen M.
Ist eine der beiden Mengen leer, so ergibt sich die leere Menge. Enthält eine der Mengen nur das leere Wort, so ergibt sich die andere Menge.
Mit M^k wird die Verkettung $\underbrace{M \circ M \circ \ldots \circ M}_{\text{k-mal } M}$ bezeichnet.

Für die Wortmenge A^* über einem Alphabet A kann kein begrenzendes k angegeben werden, denn die erzeugbaren Wörter können beliebig lang sein. Hierfür wurde der oben schon verwendete Stern-Operator eingeführt.

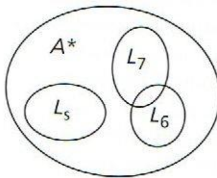
Kleene-Stern (sprich: Klini-stern) nach Stephen Cole Kleene (1909–1994)

> Der **Kleene-Stern** ist eine einstellige Operation für Mengen M, mit
> $M^* = M^0 \cup M^1 \cup M^2 \cup \ldots$

Aus der Definition ist unmittelbar erkennbar, dass
1. das leere Wort zu M^* gehört und
2. M^* eine unendliche Menge ist.

Da die Verkettung weiter oben sowohl für Zeichen als auch für Zeichenketten definiert wurde, spielt es keine Rolle, ob M ein Alphabet oder eine Menge von Wörtern ist. Auf der Basis der eingeführten Grundbegriffe kann nun der Begriff „formale Sprache" definiert werden.

5.1.6 Formale Sprache



> Eine **formale Sprache** L über einem Alphabet A ist eine beliebige Teilmenge der Wortmenge A^*. Die Elemente von L heißen **Sätze** (im Compilerbau) bzw. **Wörter** (in der Theorie der formalen Sprachen).

Es ist sofort klar, dass $L_1 = \emptyset$ (leere Sprache) und $L_2 = A^*$ (Allsprache) zulässige Sprachen über A sind. Beide erweisen sich jedoch als praktisch weitgehend uninteressant, denn wozu braucht man eine Sprache, die keinen einzigen Satz enthält, oder eine, die hinsichtlich der Satzbildung keinerlei Regeln vorgibt?

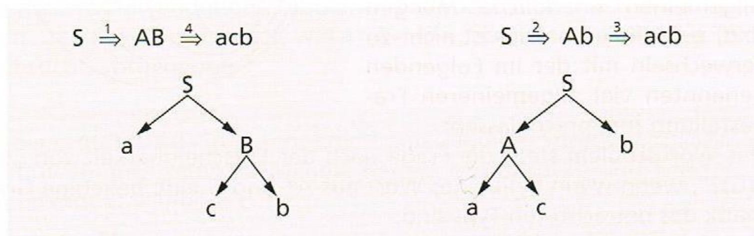
Die Fälle L(G) = Ø und L(G) = A^* sind praktisch bedeutungslos

Formale Grammatiken G sorgen dafür, dass die von ihnen erzeugbaren Sprachen $L(G)$ im Allgemeinen gerade „zwischen" Ø und A^* liegen, d. h., $\emptyset \subseteq L(G) \subseteq A^*$.
Die **von einer Grammatik G erzeugbare Sprache** ist $L(G) = \{w \mid s \stackrel{*}{\Rightarrow} w\}$, d. h., jedes Wort w, das aus dem Spitzensymbol s über beliebig viele Schritte abgeleitet werden kann, gehört zu dieser Sprache. Die Länge der Ableitung spielt keine Rolle.

- Die auf Seite 445 angegebene Grammatik G_1 beschreibt die Menge/Sprache der geraden Zahlen. T ergibt sich direkt aus A. Beispielsweise gehört die Zahl, genauer: das Zahlwort, "38" zu $L(G_1)$, denn $s = GZAHL \Rightarrow ZAHL\ GZ \Rightarrow ZIFFER\ GZ \Rightarrow 3GZ \Rightarrow$ "38".

Wenn es sich um Grammatiken handelt, deren Regeln auf der linken Seite aus genau einem Nichtterminal bestehen, spielt es keine Rolle, in welcher Reihenfolge die Nichtterminale in den Satzformen der Ableitung ersetzt werden. Sogar simultane Ersetzungen sind denkbar.
Üblicherweise werden **Linksableitungen** verwendet, d. h., das am weitesten links stehende Nichtterminal wird zuerst ersetzt. Trotz dieser Verabredung kann es vorkommen, dass mehrere (echt) verschiedene Ableitungen für ein und dasselbe Wort aus $L(G)$ existieren. Solche Grammatiken und die dazugehörigen Sprachen sind **mehrdeutig**.

- Die Grammatik
  $G_2 = (\{S, A, B\}, \{a, b, c\}, \{S \to aB \mid Ab, A \to ac, B \to cb\}, S)$
  ist mehrdeutig, denn das Wort "acb" lässt sich auf verschiedene Weise aus S linksableiten:



  Es entstehen unterschiedliche Ableitungsbäume.

Mehrdeutige Grammatiken sind höchst unerwünscht. Sie erschweren die Prozesse, die bei der Übersetzung von Programmiersprachen stattfinden, erheblich und können zu Fehlern führen.
Leider kann i. Allg. weder das Aufspüren der Mehrdeutigkeit für Grammatiken noch deren Umformung in eindeutige Grammatiken, die die gleiche Sprache beschreiben, automatisch erledigt werden. Dies bleibt im Wesentlichen „Handarbeit".

5.1.7 Chomsky-Hierarchie

Formale Grammatiken und die damit verbundenen Sprachen werden klassifiziert. Klassifizierungsmerkmal ist die Regelgestalt.

Eine formale Grammatik $G = (N, T, P, s)$ heißt

- vom **Typ 0** oder **unbeschränkt**, wenn für jede Regel $\alpha \rightarrow \beta$ gilt: $\alpha \in (N \cup T)^* \setminus T^*$ und $\beta \in (N \cup T)^*$.
- vom **Typ 1** oder **kontextsensitiv**, wenn gegenüber Typ 0 für jede Regel übereinstimmend gilt, dass die Länge von α nicht größer ist als die von β, d. h., $|\alpha| \leq |\beta|$ (Längenmonotonie). $s \rightarrow \varepsilon$ ist zulässig, wenn s auf keiner rechten Regelseite steht.
- vom **Typ 2** oder **kontextfrei**, wenn gegenüber Typ 1 für jede Regel einschränkend gilt, dass alle linken Regelseiten genau aus einem Nichtterminal bestehen, d. h., $X \rightarrow \beta$, $X \in N$. $X \rightarrow \varepsilon$ ist zulässig.
- vom **Typ 3** oder **regulär**, wenn gegenüber Typ 2 für jede Regel einschränkend $X \rightarrow \gamma$ oder $X \rightarrow \gamma Y$ gilt, wobei $\gamma \in T^*$, $X, Y \in N$.

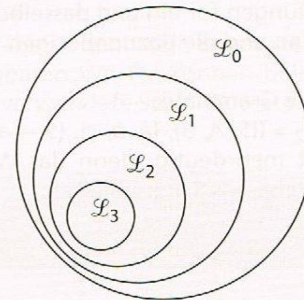
Die von den Grammatiktypen erzeugbaren Sprachen heißen dementsprechend.

Hinweis zu formalen Grammatiken des Typs 3:

Ebenso ist $X \rightarrow \gamma$ oder $X \rightarrow Y\gamma$ für alle Regeln möglich. Die Formen $X \rightarrow \gamma Y$ und $X \rightarrow Y\gamma$ dürfen jedoch nicht vermischt werden.

Die durch Grammatiken vom Typ *i* erzeugbaren Sprachklassen $\mathcal{L}_i$ (Menge von Sprachen des Typs *i*) bilden eine Hierachie, d. h., sie stehen in nebenstehend skizzierter Beziehung:

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$$



Neben den Grammatiken gibt es für jede Sprachklasse ein spezielles **Automatenmodell** und individuelle Beschreibungstechniken. Damit gelingt es, konkrete Sprachen, die im Allgemeinen *unendliche* Mengen sind, zu definieren. Dies ist nicht zu verwechseln mit der im Folgenden genannten viel allgemeineren Fragestellung für Sprachklassen:

Das **Wortproblem** stellt die Frage nach der Entscheidbarkeit von „$w \in L(G)$?", wenn *w* ein beliebiges Wort aus A^* und *G* eine beliebige Grammatik des betrachteten Typs sind.

Eine Frage oder ein Problem heißt entscheidbar, wenn es entweder mit *ja* (wahr/true) oder mit *nein* (falsch/false) beantwortet wird. Manche Problemstellungen müssen erst in eine entsprechende Entscheidungsform gebracht werden.

Satz: Das Wortproblem ist für Typ-1- und damit auch für Typ-2- und Typ-3-Sprachen entscheidbar, jedoch nicht für Typ-0-Sprachen.

Beweis:

$w = z_1 z_2 z_3 \ldots z_n$ sei ein beliebiges zu untersuchendes Wort aus A^*. Ein allgemeines Prüfverfahren für „$w \in L(G)$?" arbeitet für *G* ohne ε-Regeln folgendermaßen:

Vom Spitzensymbol *s* ausgehend werden durch Anwendung *sämtlicher* passender Regeln alle möglichen Satzformen gebildet (Simultanableitung). Mit den so gebildeten Satzformen fährt man ebenso fort,

bis deren Längen mit der von w übereinstimmen. Es gibt nur endlich viele Satzformen in $(N \cup T)^n$. Darunter befinden sich endlich viele Zeichenketten, die ausschließlich aus Terminalen bestehen. Kommt w darunter vor, gilt $w \in L(G)$ ansonsten gilt $w \notin L(G)$.
Das beschriebene Verfahren kann auf Typ-0-Grammatiken, wegen fehlender Längenmonotonie, im Allgemeinen nicht angewandt werden. Es ist nämlich möglich, dass das betrachtete Wort aus einer Satzform, die länger ist als w, ableitbar ist. Eine Begrenzung des „Suchraumes" ist daher unmöglich.

Das allgemeine Verfahren zur Entscheidung von „$w \in L(G)$?" ist sehr aufwendig und zeitraubend. Für praktische Anwendungen ist es unbrauchbar. Im Gebiet „Compilerbau" werden daher wesentlich effizientere Entscheidungsverfahren eingesetzt.

5.1.8 Reguläre Ausdrücke

Typ-3-Sprachen können außerdem durch reguläre Ausdrücke beschrieben werden. (Auf Hochkommata wird im Folgenden verzichtet.)
Reguläre Sprachen finden beispielsweise Anwendung bei
- Variablennamen (Bezeichner) in vielen Programmiersprachen: Ein **Bezeichner (identifier)** ist eine beliebige Zeichenkette über einem bestimmten Alphabet, die mit einem Buchstaben beginnen muss.
- Zahlen als Folge von Ziffern, z. B. 8364, (Problem: Unterdrückung von Vornullen).

Komplexere Anwendungen regulärer Ausdrücke finden sich in Texteditoren. Dabei werden **Muster**, wie *a*(b+c)d+* , nach denen man den Text durchsucht, vorgegeben.

Der Norton Commander unter DOS verwendet folgende Muster, um bestimmte Dateinamen zu finden: test?.doc, *.bat, [abc]*.*, [a-e]*.* [^adt]*.*

Jeder **reguläre Ausdruck** α definiert eine **reguläre Sprache** $L(\alpha)$. Man nennt sie auch **reguläre Menge**. Reguläre Ausdrücke sind wie folgend induktiv definiert:

1. $\underline{\emptyset}$ ist ein regulärer Ausdruck, und es gilt $L(\underline{\emptyset}) = \emptyset$.
2. $\underline{\varepsilon}$ ist ein regulärer Ausdruck und es gilt $L(\underline{\varepsilon}) = \{\varepsilon\}$.
3. Für jedes Alphabetzeichen $a \in A$, A ... Alphabet, ist $\underline{a}$ ein regulärer Ausdruck, und es gilt $L(\underline{a}) = \{a\}$.
4. Sind α und β reguläre Ausdrücke, so auch
   $(\alpha + \beta)$, und es gilt $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$
   $(\alpha \cdot \beta)$, und es gilt $L(\alpha\beta) = L(\alpha) \circ L(\beta)$
   (α^*), und es gilt $L(\alpha^*) = L(\alpha)^*$ ← KLEENE-Stern.

Nur die nach 1. bis 4. definierten Ausdrücke sind reguläre Ausdrücke.

Beispiele für reguläre Ausdrücke

1. $\alpha := (\underline{a} + (\underline{b}\ \underline{c}))$

$$L(\alpha) = L(\underline{a}) \cup L(\underline{b}\ \underline{c})$$
$$= L(\underline{a}) \cup L(\underline{b}) \circ L(\underline{c})$$

$= \{a\} \cup \{b\} \circ \{c\}$
$= \{a\} \cup \{bc\}$
$= \{a, bc\}$

2. $\beta := ((\underline{a} + \underline{b})\underline{c})$
   $L(\beta) = \{ac, bc\}$
3. $\gamma := ((\underline{a}^*)\underline{b})$
   $L(\gamma) = \{b, ab, a^2b, a^3b, \dots\}$

Um die Ausdrücke lesbar zu halten, wird folgende Klammersparregel eingeführt: Im Ausdruck $(\alpha\ op_1\ \beta)\ op_2\ \gamma$ dürfen die runden Klammern nicht entfallen, wenn die Priorität von op_2 größer ist als die von op_1. Die Reihenfolge der Operatoren mit aufsteigender Priorität lautet: +, · , *. Dies erinnert an das Zahlenrechnen mit + (Addition), · (Multiplikation) und ^{n} (Potenzieren), aber nicht alle Eigenschaften sind übertragbar.
Es kann gezeigt werden, dass die Menge der mit regulären Ausdrücken beschreibbaren Sprachen mit der Sprachklasse übereinstimmt, die mit Typ-3-Grammatiken definiert werden.

5.1.9 Endliche Automaten

Endliche Automaten, kurz: EA, sind eine alternative und äquivalente Beschreibungstechnik für reguläre Sprachen. Obwohl der Begriff an eine reale Maschine denken lässt, handelt es sich um ideelle Objekte, die ein bestimmtes Berechnungsmodell darstellen.

Ein **endlicher Automat (EA)** ist ein Quintupel $M = (Q, \Sigma, \delta, q_0, E)$, mit

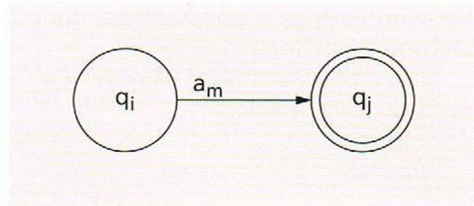
Q … endliche Menge der Zustände;
Σ … Eingabealphabet, $Q \cap \Sigma = \emptyset$;
δ … Überführungsfunktion (totale Funktion!), $Q \times \Sigma \to Q$;
q_0 … Startzustand, $q_0 \in Q$; und
E … endliche Menge der Endzustände.

Ein EA besteht aus einem Band, welches wiederum aus aufeinanderfolgenden Feldern besteht.
Das zu analysierende Eingabewort wird zeichenweise auf das Band geschrieben, wodurch eine endliche zusammenhängende Folge von Bandfeldern gefüllt wird.

Ein zum Automaten gehörender Lesekopf steht auf dem Feld, das das erste Zeichen des Eingabewortes enthält.

Jeder EA beginnt seine Arbeit im Anfangszustand q_0. In jedem Schritt oder Arbeitstakt erfolgt ein Zustandswechsel und der Lesekopf des Automaten wird um genau ein Feld (auf dem Band) nach rechts bewegt. Die Überführungsfunktion δ gibt an, welchen Folgezustand der Automat am Ende eines jeden Arbeitstakts einnimmt. Da δ eine *totale* Funktion ist, muss $\delta(q_i, a)$ für alle $q_i \in Q$ und $a \in \Sigma$ existieren. Eine zugehörige Tabelle ist also stets vollständig ausgefüllt.

δ	a_0	a_1	$\dots$	a_n
q_0	$\dots$	q_r	$\dots$	
q_1	$\dots$			
$\vdots$	$\vdots$		$\dots$	
q_m	$\dots$			



Für δ gibt es eine sehr anschauliche grafische Darstellung. Falls q_j – wie im Beispiel – ein Endzustand ist, wird dies durch doppelte Umrandung (zwei konzentrische Kreise) hervorgehoben.

Das **Akzeptanzverhalten** des Automaten lässt sich so beschreiben: Ein EA stoppt, wenn das Eingabewort vollständig abgetastet wurde und verharrt in einem Zustand q_k. Falls q_k ein Endzustand ist, wird das Eingabewort von diesem EA akzeptiert, ansonsten wird es abgewiesen.

*a*b + ab**

δ	a	b
q_0	q_2	q_1
q_1	q_5	q_5
q_2	q_3	q_6
q_3	q_3	q_4
q_4	q_5	q_5
q_5	q_5	q_5
q_6	q_5	q_6

Die vollständige Definition des EA ist
$M_1 = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \{a, b\}, \delta, q_0, \{q_1, q_2, q_4, q_6\})$.

Der Zustand q_5 wirkt wie eine Falle: Wird dieser Zustand vom Automaten erst einmal eingenommen, gibt es keinen Weg mehr hinaus. Da er selbst kein Endzustand ist, endet die Analyse mit negativem Resultat.

Die Analyse eines Eingabewortes, wie etwa "aaaab", kann durch eine **Folge von Konfigurationen** $[q_{i_1}, r_{k_1}] \vdash [q_{i_2}, r_{k_2}] \vdash [q_{i_3}, r_{k_3}] \dots$ protokolliert werden. Eine Konfiguration entspricht einem „Schnappschuss", einer Momentaufnahme der Arbeit des Automaten, und umfasst den aktuellen Zustand q_i zusammen mit dem aktuellen Restwort r_k.

Zustand	Restwort
q_0	*aaaab*
q_2	*aaab*
q_3	*aab*
q_3	*ab*
q_3	*b*
q_4	ε

$\vdash$ heißt „führt zu".

Die Abarbeitung endet in q_4. Da q_4 ein Endzustand ist, wird das Wort "aaaab" vom Automaten akzeptiert und gehört somit zu der Sprache, die der Automat beschreibt. Allgemein gilt für die **von einem EA *M* akzeptierte (erkannte, beschriebene, definierte) Sprache** *L*(*M*):

$L(M) = \{w | w \in \Sigma^* \text{ und } [q_0, w] \vdash^* [q_e, \varepsilon] \text{ und } q_e \in E\}$.

Das Symbol $\vdash^*$ bedeutet „Konfigurationsfolge beliebiger Länge".

Dies wird auch gern unter Verwendung der *erweiterten Überführungsfunktion* $\hat{\delta}$ geschrieben:

$$L(M) = \{w \mid w \in \Sigma^* \text{ und } \hat{\delta}(q_0, w) = q_e \text{ und } q_e \in E\}.$$

$\hat{\delta}$ wird rekursiv definiert:

$$\hat{\delta}(q, \varepsilon) = q$$
$$\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w)$$

5.1.10 Nichtdeterministische endliche Automaten

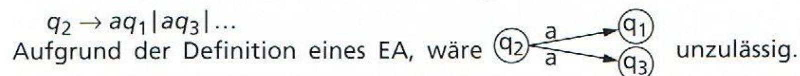
Weiter oben wurde behauptet, dass CHOMSKY-Typ-3-Grammatiken und EA äquivalente Beschreibungsmittel für reguläre Sprachen sind.

> *Satz:* Zu jeder Typ-3-Grammatik *G* gibt es einen äquivalenten EA *M* und umgekehrt.

Dies zu beweisen, bedeutet zwei Teile zu behandeln, nämlich
1. Konstruiere aus *G* einen äquivalenten EA *M* (Teil 1) und
2. Konstruiere aus *M* eine äquivalente Typ-3-Grammatik *G* (Teil 2).

Der Beweis ist konstruktiv. Teil 2 – gegeben ist *M*, gesucht ist *G*, mit $L(G) = L(M)$ – ist nicht schwer: Aus $\delta(q_i, a) = q_j$ ergibt sich die Regel $q_i \rightarrow aq_j$, wobei $q_i, q_j \in N$ und $a \in T$. Falls q_j ein Endzustand ist, kommt *zusätzlich* $q_i \rightarrow a$ hinzu. Auf diese Weise wird schließlich eine zugehörige äquivalente Grammatik aufgebaut.
In Teil 1 des Beweises, ist ein äquivalenter endlicher Automat aus einer gegebenen regulären Grammatik zu konstruieren. Dies ist problematisch, wie folgender Auszug aus irgendeiner regulären Grammatik zeigt.

$q_2 \rightarrow aq_1 \mid aq_3 \mid \ldots$

Aufgrund der Definition eines EA, wäre  unzulässig.

Wie können mehrere Regeln für q_2 mit einem einzigen Argument-Wert-Paar in der Überführungsfunktion ausgedrückt werden?
Ein kleiner Trick ist erforderlich: Alle möglichen Folgezustände werden zu einer Menge zusammengefasst. Dann ist obige Darstellung durchaus berechtigt, denn der Funktionswert ist in diesem Beispiel $\delta(q_2, a) = \{q_1, q_3\}$.

δ	…	a	…
⋮	⋮	⋮	⋮
q_2	…	$\{q_1, q_3\}$	…
⋮	⋮	⋮	⋮

Im Tabellenkörper stehen also Mengen, genauer: Teilmengen von *Q*. Die Menge aller dieser Teilmengen von *Q* ist die Potenzmenge *P*(*Q*) von *Q*. Sie besitzt $2^{|Q|}$ Elemente, denn *Q* ist endlich.

Diese Modifikation des endlichen Automaten erfordert eine begriffliche Unterscheidung von der Basisform, die nachträglich **deterministischer endlicher Automat**, kurz: EA oder **DEA**, genannt wird. Der in diesem Abschnitt entwickelte Automat heißt **nichtdeterministischer endlicher Automat**, kurz: **NEA**.

Ein **nichtdeterministischer endlicher Automat (NEA)** wird durch ein Quintupel $M = (Q, \Sigma, \delta, q_0, E)$ definiert, wobei bis auf δ die Bedeutungen der Symbole aus der Definition des (D)EA übernommen werden. Die Überführungsfunktion eines NEA ist wie folgt definiert: $\delta : Q \times \Sigma \to P(Q)$. δ ist eine totale Funktion.

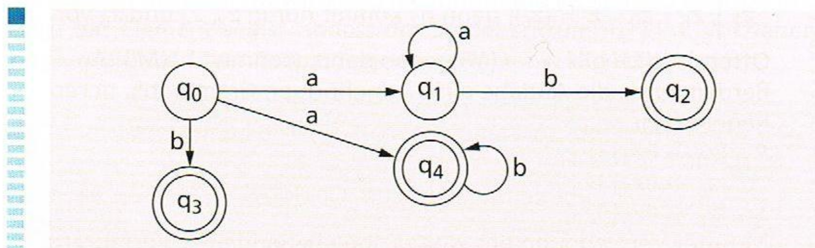
Da $\emptyset \in P(Q)$ für jede beliebige Menge Q gilt, kann es auch $\delta(q, a) = \emptyset$ geben. Dies wird als verbotener Übergang interpretiert. Der damit verbundene Crash bedeutet Misserfolg des betrachteten Automaten bei dessen Analysetätigkeit.

Wie arbeitet ein NEA?
Falls es in einem Zustand, wie oben in q_2, mit ein und demselben Zeichen a des Eingabealphabets Σ mehr als einen Folgezustand gibt, z.B. q_1, q_3, so wird der Automat entsprechend oft kopiert (geklont). Jeder, der so entstandenen „ununterscheidbaren Zwillingsautomaten" wird in je einem der beiden Folgezustände mit dem aktuellen Band gestartet. Die Arbeit der Zwillingsautomaten erfolgt zeitlich parallel. Sobald einer von ihnen einen Erfolg signalisiert, wird die Arbeit der anderen eingestellt. Das Eingabewort gilt als akzeptiert. Anderenfalls müssen sie solange arbeiten, bis schließlich jeder von ihnen stoppt und einen Misserfolg gemeldet hat. Erst dann kann man sagen, dass das Eingabewort nicht zu der durch den Automaten festgelegten Sprache gehört.

Formal ausgedrückt bedeutet das:
$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap E \neq \emptyset\}$

Unter Verwendung von NEA können reguläre Sprachen im Allgemeinen kompakter definiert werden als mit DEA.

Zum direkten Vergleich mit dem weiter oben angegebenen DEA wird im nächsten Beispiel ein NEA für $\underline{a}^*\underline{b} + \underline{a}\,\underline{b}^*$ definiert.



Anstelle zu zeigen, dass es zu jeder regulären Grammatik einen äquivalenten DEA gibt, entstand ein NEA, der zur Beschreibung von Sprachen offenbar gute Dienste leistet, die eigentliche Beweisaufgabe ist damit jedoch nicht gelöst. Es bleibt zu zeigen, dass zu jedem NEA ein äquivalenter DEA konstruiert werden kann. Dies bedeutet dann außerdem, dass die Leistungsfähigkeit von DEA und NEA bei der Beschreibung regulärer Sprachen übereinstimmt.

Die Beweisidee liegt darin, den oben angewandten Trick rückgängig zu machen, indem jeder Menge aus $P(Q)$ ein Zustand z_i zugeordnet wird. Dann kann die Überführungsfunktion δ' des gesuchten DEA aus δ des betrachteten NEA $M = (Q, \Sigma, \delta, q_0, E)$ nach der Vorschrift

$$\delta'(z_i, a) = \bigcup_{q \in z_i} \delta(q, a)$$

konstruiert werden.

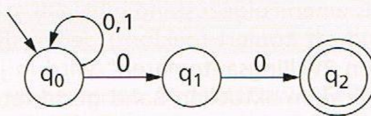
Die anderen Bestandteile des DEA $M' = (Q', \Sigma', \delta', z'_0, E')$ ergeben sich aus

$Q' = P(Q)$

$z'_0 = \{q_0\}$

$E' = \{z_i \in Q' \mid z_i \subseteq Q \text{ und } z_i \cap E \neq \emptyset\}$.

■ Konstruktion eines äquivalenten DEA aus einem NEA



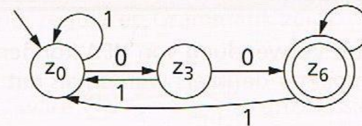
δ in Tabellenform

δ	0	1
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\emptyset$

$$P(Q) = \{\underbrace{\{q_0\}}_{z_0}, \underbrace{\{q_1\}}_{z_1}, \underbrace{\{q_2\}}_{z_2}, \underbrace{\{q_0, q_1\}}_{z_3}, \underbrace{\{q_0, q_2\}}_{z_4}, \underbrace{\{q_1, q_2\}}_{z_5}, \underbrace{\{q_0, q_1, q_2\}}_{z_6}, \underbrace{\emptyset}_{z_7}\}$$

z. B. $\delta'(z_3, 0) = \delta'(\{q_0, q_1\}, 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$
$= \{q_0, q_1\} \cup \{q_2\} = \{q_0, q_1, q_2\}$
$= z_6$

Das Ergebnis lautet:



δ'	0	1
z_0	z_3	z_0
z_1	z_6	z_0
z_2	z_6	z_0

$Q' = \{z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7\}$
$z' = z_0$, denn q_0 ist der Anfangszustand von M.
$E' = \{z_6\}$, denn q_2 kommt nur in z_2, z_4 und z_6 vor.

Offensichtlich gilt $w \in L(M)$ genau dann, wenn $w \in L(M')$. Außerdem wäre die Angabe einer zugehörigen Grammatik, deren Regelmenge

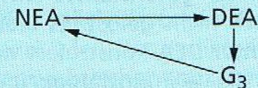
$P = \{z_0 \to 0z_3 \mid 1z_0$
$z_3 \to 0z_6 \mid 1z_0$
$z_6 \to 0z_6 \mid 1z_0\}$

ist unter Verwendung des weiter oben behandelten Verfahrens unmittelbar möglich.

DEA: deterministischer endlicher Automat

NEA: nichtdeterministischer endlicher Automat

Zusammenfassung:
Reguläre Ausdrücke, Typ-3-Grammatiken, DEA und NEA sind äquivalente Beschreibungsmittel regulärer Sprachen.



5.1.11 Kellerautomaten und kontextfreie Sprachen

Endliche Automaten definieren lediglich reguläre Sprachen. Um kontextfreie Sprachen beschreiben zu können, muss dieser Automatentyp erweitert werden. Welche Minimal-Erweiterungen nötig sind, sollen die folgenden Beispiele auszuloten helfen.

- Beispiel 1:
  Die Sprache der Palindrome über $\{a, b\}$. Palindrome sind Wörter, die mit ihrem jeweiligen Umkehrwort übereinstimmen, etwa *abba* und *bbabb*.

- Beispiel 2:
  Die Sprache $L = \{a^n b^n \mid n \geq 1\}$. Sie wird von der Grammatik G, mit $G = (\{S\}, \{a, b\}, \{S \rightarrow ab \mid aSb\}, S)$ beschrieben.

- Beispiel 3:
  Die Sprache der korrekt geklammerten arithmetischen Ausdrücke über dem Alphabet $\{a, (,), +, *\}$.

Zur Analyse von Wörtern in diesen Beispielsprachen wird ein Zusatzspeicher benötigt, der zum „Merken" des jeweils ersten Teils des Wortes (Beispiel 1) bzw. der eingelesenen *a*'s (Beispiel 2) bzw. des Anfangsstückes eines Klammerausdrucks (Beispiel 3) dient. Ein einfaches „Mitzählen" der eingelesenen Zeichen – etwa in Beispiel 1 – würde nicht ausreichen, denn auch deren Reihenfolge muss beachtet werden.

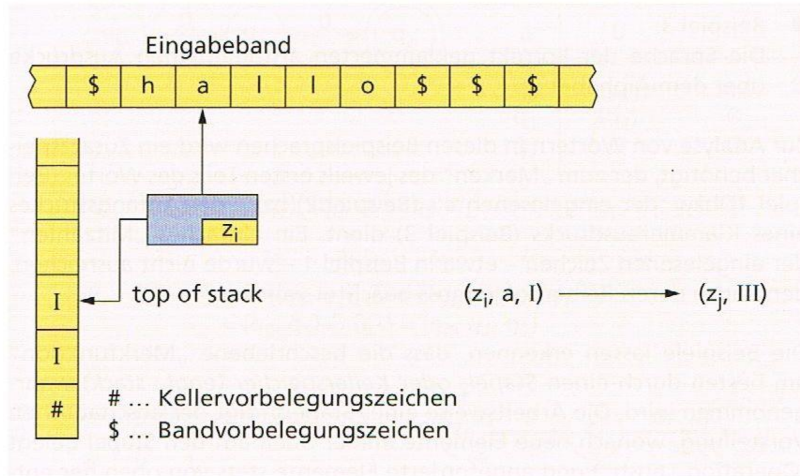
Die Beispiele lassen erkennen, dass die beschriebene „Merkfunktion" am besten durch einen *Stapel- oder Kellerspeicher* (*engl.: stack*) wahrgenommen wird. Die Arbeitsweise eines Stapels folgt der anschaulichen Vorstellung, wonach neue Elemente immer oben auf den Stapel gelegt (Operation „push") und angeforderte Elemente stets von oben her entnommen (Operation „pop") werden. Das aktuell oberste Stapelement wird „top of stack" genannt. Zu beachten ist, dass pop das top of stack vom Stapel entfernt. Es wird also nicht nur eine Wertkopie benutzt, sondern das Element selbst, sodass der Stapel schrumpft. Ein „Vorziehen" eines mitten auf dem Stapel liegenden Elements ist verboten.

Ein **nichtdeterministischer Kellerautomat** ist definiert durch ein 7-Tupel

$$M = (Z, \Sigma, \Gamma, \delta, z_0, k_0, E), \text{ mit}$$

Z … endliche Menge der Zustände;
Σ … Eingabealphabet;
Γ … Kelleralphabet;
δ … partielle Überführungsfunktion:
$Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow P_{\text{endlich}}(Z \times \Gamma^*)$;
z_0 … Anfangszustand, $z_0 \in Z$;
k_0 … Kellervorbelegungszeichen, $k_0 \in \Gamma$;
E … Menge der Endzustände.

Arbeitsweise:
In jedem Arbeitstakt liest der Automat das top of stack *A verbrauchend* vom Stapel. Der aktuelle Zustand z_i, das aktuelle Zeichen *a* auf dem Eingabeband und *A* bilden die Argumente der Überführungsfunktion δ.
Falls δ für das spezielle Tripel (z_i, a, A) definiert ist, bestimmt der Funktionswert $\delta(z_i, a, A) \ni (z_j, s_1 s_2 s_3 \ldots s_r)$ den Folgezustand z_j und das auf den Stapel zu schreibende Kellerwort $s_1 s_2 s_3 \ldots s_r$, wobei die Speicherung mit s_r, s_{r-1} beginnt, sodass schließlich s_1 das neue top of stack ist. Das zu speichernde Kellerwort darf leer sein.
Wegen $Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ sind für δ Argumente der Gestalt (z_i, ε, A) erlaubt. Dieser Spezialfall (ε an zweiter Stelle) bedeutet einen *spontanen Übergang*, also ein Arbeitstakt, bei dem der Lesekopf auf dem aktuellen Feld des Eingabebandes verharrt und dessen Inhalt unbeachtet lässt.
Anderenfalls, d.h., wenn $a \neq \varepsilon$, wird *a* gelesen und der Kopf rückt am Ende dieses Arbeitstaktes um genau ein Feld nach rechts.



Akzeptanzverhalten:
Der Automat stoppt erfolgreich, wenn das Eingabewort vollständig abgetastet (gescannt) wurde *und* der dann aktuelle Zustand ein Endzustand ist. Der Kellerinhalt spielt *keine* Rolle. In diesem Falle wird gesagt, dass der Automat das Eingabewort akzeptiert hat.

↗ auch S. 453

Die Menge aller Wörter, die von einem speziellen Kellerautomaten *M* akzeptiert werden, ist die **von *M* akzeptierte Sprache** $L(M)$.
$L(M) = \{w \mid w \in \Sigma^* \text{ und } \hat{\delta}(z_0, w, k_0) \cap E \neq \emptyset\}$

Die Definition der erweiterten Überführungsfunktion erfolgt sinngemäß wie bei endlichen Automaten. $L(M)$ kann auch unter Verwendung der Konfigurationenfolge definiert werden.

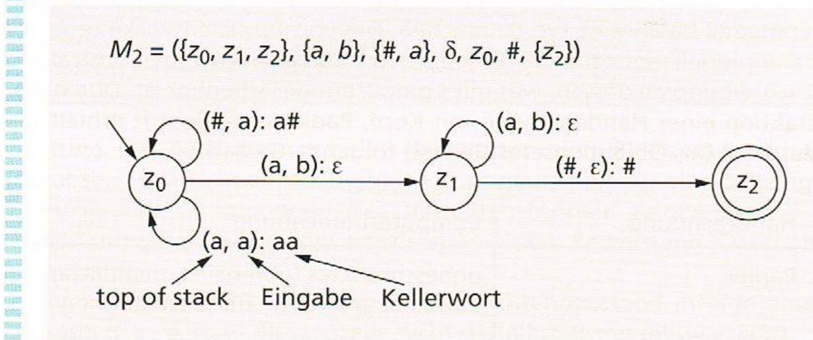
↗ auch S. 450, 451

$$L(M) = \{w \mid w \in \Sigma^* \text{ und } [z_0, w, k_0] \vdash^* [z_e, \varepsilon, \alpha] \text{ und } z_e \in E, \alpha \in \Gamma^*\}$$

Bei der Definition des Konfigurationenübergangs kann man sich ebenfalls an der für endliche Automaten orientieren.

Eine undefinierte Stelle der Überführungsfunktion führt zum erfolglosen Abbruch (Crash) der Arbeit des geklonten Automaten. Es gibt also zwei Möglichkeiten für $w \notin L(M)$, nämlich Crash und $z_e \notin E$ nach vollständigem Lesen des Eingabewortes. In allen Fällen stoppt der Automat.
Nichtdeterminismus bei abstrakten Automaten wurde bereits für NEA erläutert. Dies gilt sinngemäß für nichtdeterministische Kellerautomaten.
Der folgende nichtdeterministische Kellerautomat beschreibt die im Beispiel 2 auf S. 455 vorgestellte Sprache $L(M) = \{a^n b^n \mid n \geq N\}$.

Der nichtdeterministische Kellerautomat M_2 ist wie folgt definiert.



Leider kann δ nicht als Tabelle dargestellt werden, so wie dies bei endlichen Automaten möglich war. Eine dritte Dimension wäre nötig, was die Übersichtlichkeit auf Papier negativ beeinträchtigen würde. Insofern ist die grafische Darstellung von δ klar zu bevorzugen.
M_2 akzeptiert beispielsweise das Wort *aabb*. Die folgende Tabelle zeigt die schrittweise Abarbeitung. Jede Zeile in der Tabelle charakterisiert eine *Konfiguration*. Die Folge der Konfigurationen beginnt stets mit $[z_0, w, \#]$ und endet bei Akzeptanz von w mit $[z_e, \varepsilon, \alpha]$, wobei $z_e \in E$ gilt und α irgendein Wort auf dem Keller ist.

Der nichtdeterministische Kellerautomat im obigen Beispiel arbeitet quasi deterministisch. Die dort angegebene Überführungsfunktion δ gibt für jedes top-of-stack-Eingabezeichen-Paar genau ein Folgezustand-Kellerwort-Paar an. Ein Klonen des betreffenden Automaten kann also nicht vorkommen.

Zustand	Eingabe-(rest)wort	top ↓ Keller
z_0	*aabb*	#
z_0	*abb*	*a*#
z_0	*bb*	*aa*#
z_1	*b*	*a*#
z_1	ε	#
z_2	ε	#

Man kann zeigen, dass die Klasse der kontextfreien Sprachen durch nichtdeterministische Kellerautomaten vollständig erfasst wird.
Deterministische Kellerautomaten verfügen nicht über die „Reichweite" nichtdeterministischer Kellerautomaten. Sie definieren die sogenannten **deterministischen kontextfreien Sprachen**, die eine echte Teilmenge der kontextfreien Sprachen bilden. Sie sind für die Praxis (Programmiersprachen, Compilerbau) die bedeutsamste Klasse und beinahe so mächtig wie kontextfreie Sprachen.