Spezifikation formaler Sprachen

Hans Jürgen Ohlbach

Keywords: Sprachen, Grammatiken, Chomsky-Hierarchie, Wortproblem, Syntaxbaum

Empfohlene Vorkenntnisse: Funktionen und Relationen

Inhaltsverzeichnis

1 Motivation

Für Computer ist es extrem schwierig, natürlichsprachliche Texte zu verstehen, und damit sinnvoll etwas anzufangen. Es ist sehr viel einfacher, wenn die Texte, die ein Computer einlesen soll, eine ganz bestimmte Struktur aufweisen, die entsprechende Programme kennen, so dass sie dann die Texte sinnvoll verarbeiten können. Ein Beispiel sind Programme in einer vorgegebenen Programmiersprache, die ein entsprechender Compiler in Maschinensprache übersetzen muss, die man dann ausführen kann. Die Struktur der Programmiersprache muss exakt definiert sein, damit der Compiler die Bedeutung der einzelnen Anweisung erkennen kann.

Für die Definition einer formalen Sprache braucht man selbst wiederum einen formalen Mechanismus. Davon gibt es allerdings sehr unterschiedliche Ansätze.

Zunächst einmal muss man allerdings festlegen, was eine formale Sprache überhaupt ist: Eine formale Sprache ist eine Teilmenge aller Wörter, die sich aus einem bestimmten Alphabet bilden lassen.

Definition 1 (Alphabet und Sprache)
Ein Alphabet Σ *ist eine* endliche nichtleere *Menge, (auch* Signatur *genannt).*

Für ein Alphabet Σ *ist* Σ^* *die Menge aller endlichen Wörter (Zeichenketten), die sich mit den Elementen aus* Σ *bilden lassen.* Σ^* *enthält immer unendlich viele Elemente.*

Eine formale Sprache L *über einem Alphabet* Σ *ist eine Teilmenge von* Σ^*. L *kann leer sein, endliche viele Element enthalten, oder unendlich viele Elemente enthalten.*

Beispiele:
Für $\Sigma = \{0, 1\}$ ist Σ^* die Menge aller endlichen Bitfolgen. Eine formale Sprache L über Σ könnte z.B. die Menge aller endlichen Bitfolgen, die mit 0 enden sein. Das wären alle geraden Zahlen in Binärdarstellung.

Für $\Sigma = \{0, \ldots, 9\}$ ist Σ^* die Menge aller natürlichen Zahlen in Dezimaldarstellung (mit führenden Nullen). Eine formale Sprache über Σ könnte z.B. die Menge aller Primzahlen in Dezimaldarstellung sein.

Für $\Sigma = \{a, \ldots, z\}$ ist Σ^* die Menge aller Zeichenketten aus Kleinbuchstaben. Eine formale Sprache über Σ könnte die Menge aller *deutschen Wörter* aus Kleinbuchstaben sein.

Für $\Sigma =$ alle Unicode Zeichen ist Σ^* die Menge aller Zeichenketten aus Unicode-Zeichen. Eine formale Sprache über Σ könnte die Menge aller syntaktisch korrekten Java Programme sein. (In diesem Fall würde man ein konkretes Java Programm ebenfalls als *Wort* der Sprache bezeichnen).

Die Menge Σ muss nicht unbedingt Buchstaben aus bekannten Alphabeten enthalten. Sie kann im Prinzip beliebige Objekte enthalten, z.B. könnte Σ die Menge aller Menschen sein. Σ^* wäre dann die Menge aller endlichen Menschenketten. Eine formale Sprache über Σ könnte die Menge aller endlichen Menschenketten aus Frauen sein.

In diesen Beispielen haben wir eine Sprache über Σ durch eine natürlichsprachliche Beschreibung

spezifiziert. Eine etwas präzisere Beschreibung ermöglicht die *mathematische Mengennotation*.

Beispiele:
Für $\Sigma = \{0, 1\}$ sei L = $\{a_1 \dots a_n 0 \mid a_i \in \Sigma\}$ die Menge der geraden Zahlen in Binärdarstellung.

Für $\Sigma = \{a, b, c\}$ sei L = $\{a^n b^n b^n \mid n > 0\} = \{abc, aabbcc, aaabbbccc, \dots\}$

Folgende Notationen sind dabei sehr nützlich:
Seien X, Y *Mengen* von Wörtern über einem Alphabet.
In den Beispielen seien $X = \{ab, cd\}$ und $Y = \{0, 1\}$

Operator	Bedeutung
X^n	Die Elemente von X wird n mal hintereinander angehängt. $X^2 = \{abab, abcd, cdcd, cdab\}$
X^*	Die Elemente von X werden beliebig oft (incl. 0 mal) hintereinander angehängt. $X^* = \{\epsilon, ab, cd, abab, cdcd, abcd, cdab, \dots\}$
X^+	X wird beliebig oft, aber mindestens einmal hintereinander angehängt. $\{ab, cd, abab, cdcd, abcd, cdab, \dots\}$
$X?$	X wird null oder einmal hintereinander angehängt. $X? = \{\epsilon, ab, cd\}$
XY	steht für die Konkatenation (Hintereinanderhängung) von X und Y $XY = \{ab0, ab1, cd0, cd1\}$

Dabei steht ϵ für das leere Wort (den leeren String).

Mit diesen Hilfsmitteln kann man z.B. die Menge der geraden Zahlen in Binärdarstellung einfach als $\Sigma^*\{0\}$ mit $\Sigma = \{0, 1\}$ schreiben.

Ein häufig vorkommendes Problem ist es, herauszufinden, ob ein Wort in einer gegebene Sprache L ist, oder nicht.

Definition 2 (Das Wortproblem)
Für eine Sprache L über einem Alphabet Σ ist das Wortproblem *das Problem, herauszufinden, ob ein gegebenes Wort w aus Σ^* in L ist, oder nicht.*
D.h. gilt $w \in L$, oder gilt $w \notin L$?

Ist die Sprache L endlich, ist das Wortproblem ganz einfach zu lösen:
man geht alle Wörter in L durch, und vergleicht sie mit w.

Ist die Sprache jedoch unendlich, dann braucht man einerseits einen endlichen Beschreibungsmechanismus für die Sprache, und andrerseits einen Algorithmus, der anhand dieser Beschreibung das Wort testet.

Betrachten wir als Beispiel die Sprache $L = \{a^n b^n | n > 0\}$. Ein Algorithmus, der z.B. herausfindet, dass das Wort $aabbb$ nicht in der Sprache ist, muss testen, ob das Wort aus einer Sequenz von as gefolgt von einer gleichlangen Sequenz von bs besteht.

1.1 Parsing

Mit die wichtigsten Sprachen in der Informatik sind die Programmiersprachen. Hier manifestiert sich das Wortproblem darin, herauszufinden, ob ein gegebener Text ein syntaktisch korrektes Programm darstellt, oder nicht. Daher sagt man nicht, „der Text ist ein Element der Programmiersprache“, sondern einfacher „das Programm ist syntaktisch korrekt“. Es ist aber dasselbe gemeint.

Den Test auf syntaktische Korrektheit machen i.A. die *Parserkomponenten* eines Compilers. Ein Parser testet zum einen, ob eine Zeichenkette syntaktisch korrekt ist, bei Programmen heißt das z.B. ob alle Klammern stimmen, alle Kommas und Semikolons richtig sitzen, und noch vieles mehr. Ein Parser sollte aber nicht nur das Wortproblem lösen. Falls das Programm *nicht* syntaktisch korrekt ist, sollte er die Stellen herausfinden und anzeigen, wo der Text fehlerhaft ist, damit man ihn gezielt korrigieren kann.

Falls das Programm syntaktisch korrekt ist, sollte der Parser als Ergebnis nicht nur „korrekt“ ausgeben. Er sollte zusätzlich eine Datenstruktur erzeugen, die *die Struktur* des Programms wiedergibt. Diese Struktur ist dann die Eingabe für weitere Komponenten eines Compilers, insbesondere den Code-Generator. Eine geeignete Datenstruktur dafür ist der *Syntaxbaum*, auf den wir in Kap. 4 eingehen.

2 Grammatiken

Die mathematische Schreibweise ist kurz und prägnant – für Menschen. Für Computer ist sie aber immer noch nicht gut geeignet. Daher hat man weitere Spezifikationsmethoden erfunden, die von Computern besser verarbeitet werden können. Eine davon sind die *Grammatiken*.

Grammatiken kennt man aus dem Deutsch- oder Englischunterricht in der Schule. Da erfährt man z.B. „Ein Satz besteht aus Subjekt, Prädikat und Objekt“. „Ein Subjekt besteht aus Artikel, Attribut und Substantiv“ usw. Dieses Beispiel, etwas erweitert, kann man als sog. *Produktionssystem* mit *Produktionsregeln* schreiben:

Satz	$\rightarrow$	Subjekt Prädikat Objekt	Adjektiv	$\rightarrow$	kleine
Subjekt	$\rightarrow$	Artikel Attribut Substantiv	Adjektiv	$\rightarrow$	bissige
Artikel	$\rightarrow$	ϵ (das leere Wort)	Adjektiv	$\rightarrow$	große
Artikel	$\rightarrow$	der	Substantiv	$\rightarrow$	hund
Artikel	$\rightarrow$	die	Substantiv	$\rightarrow$	katze
Artikel	$\rightarrow$	das	Prädikat	$\rightarrow$	jagt
Attribut	$\rightarrow$	ϵ	Objekt	$\rightarrow$	Artikel Attribut Substantiv
Attribut	$\rightarrow$	Adjektiv Attribut			

In dieser Notation unterscheidet man *Variablen*, oder sog. *Nicht-Terminalsymbole* und *Konstanten* oder *Terminalsymbole*. Die Variablen sind Satz, Subjekt, Prädikat, Objekt, Artikel, Attribut, Substantiv. Die Konstanten sind der, die, das, kleine, bissige, große, hund, katze, jagt. Variablen sind nicht Bestandteil der eigentlichen Sprache, sondern müssen sukzessive durch Teilwörter ersetzt werden.

Die Produktionsregeln kann man jetzt schrittweise auf die *Startvariable* Satz anwenden, und erhält

dann z.B. folgende Ableitung

Ableitung	Regel
Satz$\Rightarrow$ Subjekt Prädikat Objekt	Subjekt $\rightarrow$ Artikel Attribut Substantiv
$\Rightarrow$ Artikel Attribut Substantiv Prädikat Objekt	Artikel $\rightarrow$ der
$\Rightarrow$ der Attribut Substantiv Prädikat Objekt	Attribut $\rightarrow$ Adjektiv Attribut
$\Rightarrow$ der Adjektiv Attribut Substantiv Prädikat Objekt	Adjektiv $\rightarrow$ kleine
$\Rightarrow$ der kleine Attribut Substantiv Prädikat Objekt	Attribut $\rightarrow$ Adjektiv Attribut
$\Rightarrow$ der kleine Adjektiv Attribut Substantiv Prädikat Objekt	Adjektiv $\rightarrow$ bissige
$\Rightarrow$ der kleine bissige Attribut Substantiv Prädikat Objekt	Attribut $\rightarrow \epsilon$
$\Rightarrow$ der kleine bissige Substantiv Prädikat Objekt	Substantiv$\rightarrow$ hund
$\Rightarrow$ der kleine bissige hund Prädikat Objekt	Prädikat $\rightarrow$ jagt
$\Rightarrow$ der kleine bissige hund jagt Objekt	Objekt $\rightarrow$ Artikel Attribut Substantiv
$\Rightarrow$ der kleine bissige hund jagt Artikel Attribut Substantiv	Artikel $\rightarrow$ die
$\Rightarrow$ der kleine bissige hund jagt die Attribut Substantiv	Attribut $\rightarrow$ Adjektiv Attribut
$\Rightarrow$ der kleine bissige hund jagt die Adjektiv Attribut Substantiv	Adjektiv $\rightarrow$ große
$\Rightarrow$ der kleine bissige hund jagt die große Attribut Substantiv	Attribut $\rightarrow \epsilon$
$\Rightarrow$ der kleine bissige hund jagt die große Substantiv	Substantiv$\rightarrow$ katze
$\Rightarrow$ der kleine bissige hund jagt die große katze	

Wählt man die anzuwendenden Regeln anders aus, erhält man andere Sätze.

Bemerkung: Was man in diesem natürlichsprachlichen Beispiel als *Satz* bezeichnet, bezeichnet man in der Theorie der Formalen Sprachen als *Wort*, bestehend aus den Signaturelementen (Konstanten, Buchstaben) der, die, das, kleine, bissige, große, hund, katze, jagt.

Eine Grammatik kann man jetzt präzise folgendermaßen definieren:

Definition 3 (Grammatik)
Eine Grammatik $G = (V, \Sigma, P, S)$ besteht aus

V : *einer Menge von Variablen (Nicht-Terminalsymbole)*

Σ : *einer Menge von Konstanten (Terminalsymbole)*

P : *einer Menge von Produktionsregeln der Gestalt $linkeSeite \rightarrow rechteSeite$*
wobei linke und rechte Seite Folgen von Variablen und Konstanten sind.
Die linke Seite darf aber nicht nur aus Konstanten bestehen.

S : *einer Startvariablen.*

Die Ersetzungen, wie im obigen Beispiel illustriert, beginnend mit der Startvariablen, erzeugen nacheinander Folgen von Terminal- und Nicht-Terminalsymbolen. Erst wenn es nur noch Terminalsymbole gibt, endet die Sequenz von Ersetzungen. Diese Folge von Ersetzungen definiert eine *Ableitungsrelation*.

Definition 4 (Ableitungsrelation, Sprache)
Für eine Grammatik $G = (V, \Sigma, P, S)$, definieren wir eine Ableitungsrelation

$$u \Rightarrow_G v$$

wobei u und v Zeichenketten aus $(V \cup \Sigma)^*$ sind, indem in u die linke Seite einer Regel aus P durch die rechte Seite ersetzt wird.

$\Rightarrow_G^*$ *ist die* transitive Hülle *von* $\Rightarrow_G$.
$u \Rightarrow_G^* v$ *bedeutet: v entsteht aus u durch eine endliche Folge von Ersetzungen.*

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$$

ist die von G erzeugte Sprache.
Die Wörter als $L(G)$ bestehen damit nur aus Konstanten.

Im obigen Beispiel ist eine der Ableitungen:

	der kleine bissige hund Prädikat Objekt	Prädikat → jagt
	↓	
$\Rightarrow_G$	der kleine bissige hund jagt Objekt	

Weiterhin gilt für das Beispiel:
Satz $\Rightarrow_G^*$ der kleine bissige hund jagt die große katze

Die von der Grammatik erzeugte Sprache $L(G)$ besteht aus allen Sätzen, die sich auf diese Weise aus der Startvariablen S ableiten lassen, und die nur aus Konstanten bestehen.

Wenn es klar ist, welche Grammatik gemeint ist, kann man in der Ableitungsrelation auch die Grammatik weglassen und nur $\Rightarrow$ statt $\Rightarrow_G$ schreiben.

Extended Backus-Naur Form:
Grammatiken in der oben eingeführten Schreibweise sind oft sehr umständlich aufzuschreiben, so dass man eine Kurzschreibweise eingeführt hat:

- anstelle von $A \to \beta_1, \ldots, A \to \beta_n$ schreibt man $A \to \beta_1 \mid \ldots \mid \beta_n$ (| steht für *oder*).
- anstelle von $A \to \alpha\gamma, A \to \alpha\beta\gamma$ schreibt man $A \to \alpha[\beta]\gamma$
  (β kann, muss aber nicht eingefügt werden)
- anstelle von $A \to \alpha\gamma, A \to \alpha B\gamma, B \to \beta, B \to B\beta$ schreibt man $A \to \alpha\{\beta\}\gamma$
  (β kann beliebig oft, auch 0 mal, eingefügt werden)

Die folgende Grammatik beschreibt arithmetische Ausdrücke über den arithmetischen Variablen x, y, z und den Zahlen 0,...,9.

$G = (\{A\}, \{(,), x, y, z, 0, \ldots, 9, +, *\}, P, A\}$
mit

$$\begin{aligned} P \;=\; & \{ \\ & A \to x \mid y \mid z \mid 0 \mid \ldots \mid 9 \\ & A \to (A + A) \mid (A * A) \\ & \} \end{aligned}$$

Die Konstanten (Terminalsymbole) sind $(,), x, y, z, 0, \ldots, 9, +, *$. Es gibt nur eine Variable (Nicht-Terminalsymbol), nämlich A.

Mit dieser Grammatik kann man z.B. folgenden arithmetischen Ausdruck erzeugen:

$$A \Rightarrow (A + A) \Rightarrow ((A * A) + A) \Rightarrow ((x * A) + A) \Rightarrow ((x * y) + A) \Rightarrow ((x * y) + 3)$$

Alle brauchbaren Grammatiken sind *nicht-deterministisch*, d.h. es sind bei den Ableitungen in den Zwischenergebnissen, mehrere Regeln anwendbar. Nur so kann man auch mehrere Wörter erzeugen. Wäre immer nur eine Regel anwendbar, dann würde auch nur ein einziges Wort erzeugt werden. Dann könnte man das Wort auch direkt hinschreiben, und bräuchte keine Grammatik.

3 Die Chomsky Hierarchie

Einer der berühmtesten Wissenschaftler, *Noam Chomsky* hat die Grammatiken nach ihrer Struktur in 4 Typen eingeteilt, der *Chomsky Hierarchie*.

Typ 0: alle Grammatiken

Typ 1 (kontextsensitiv): für alle Produktionsrelgen der Gestalt $l \to r$ gilt $|l| \leq |r|$,
wobei für ein Wort w, $|w|$ seine Länge angibt.
D.h. die Regeln verkürzen keine Wörter

Typ 2 (kontextfrei): die linke Seite aller Regeln ist eine Variable.

Typ 3 (regulär): zusätzlich gilt noch: Die rechte Seite der Regeln ist eine Konstante oder eine Konstante gefolgt von einer Variablen.
Die Regeln haben also die Gestalt:
$A \to a$ oder $A \to aB$,
wobei A und B Variablen sind, und a eine Konstante.

Mit dieser Definition gilt: Typ 3 $\subseteq$ Typ 2 $\subseteq$ Typ 1 $\subseteq$ Typ 0. Man bezeichnet daher Typ 3 als *spezieller* als Typ 2, und dieser ist spezieller als Typ 1, welcher wiederum spezieller als Typ 0 ist.

Der Typ einer Grammatik besagt noch nicht unbedingt etwas über die Eigenschaften der von ihr generierten Sprache. So wie man ein Programm umständlicher und ineffizienter als ein anderes Programm

schreiben kann, welches das gleiche berechnet, so kann man auch Grammatiken umständlicher schreiben als nötig.

Beispiel: Die folgende Produktionsregel ist vom Typ 2: $V \to abcdW$.
Durch Einführung von Hilfsvariablen kann man sie in Typ 3-Produktionen umwandeln, die aber genau die gleichen Wörter produzieren:
$V \to aW_1$
$W_1 \to bW_2$
$W_2 \to cW_3$
$W_3 \to dW$

Das bedeutet, unter Umständen ist es möglich, eine Grammatik eines bestimmten Typs in eine andere Grammatik eines spezielleren Typs umzuwandeln, die aber genau die gleiche Sprache produziert.

Dies motiviert die folgende Definition:

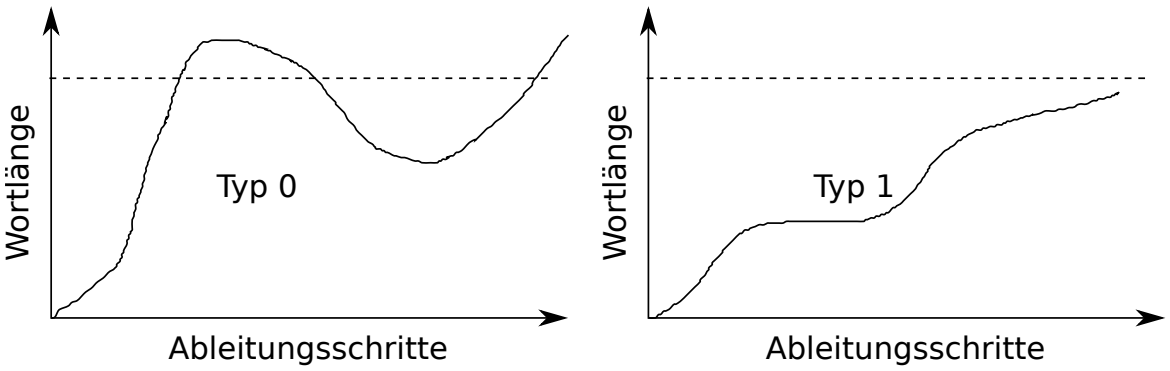
Definition 5 (Typ einer Sprache)
Der Typ einer Sprache ist der speziellste Typ der Grammatik, die diese Sprache erzeugt.

Beispiele für Sprachen der verschiedenen Typen:
$L = \{a^n b^m \mid n, m > 0\}$ ist vom Typ 3.
$L = \{a^n b^n \mid n > 0\}$ ist vom Typ 2, aber nicht vom Typ 3
$L = \{a^n b^n c^n \mid n > 0\}$ ist vom Typ 1, aber nicht vom Typ 2
$L =$ terminierende Programme in einer höheren Programmiersprache ist vom Typ 0, nicht vom Typ 1
$L =$ nicht-terminierende Programme in einer höheren Programmiersprache ist nicht vom Typ 0.

Diese Typhierarchie erscheint auf den ersten Blick sehr künstlich. Wir werden aber sehen, dass die vier Typen sehr unterschiedliche Eigenschaften haben. Für Zwecke der Informatik sind eigentlich nur Typ 3 und Typ 2 interessant, einerseits weil für sie das Wortproblem noch effizient lösbar ist, und andrerseits weil man nur aus deren Wörtern einen Syntaxbaum erzeugen kann, der die Basis für eine Weiterverarbeitung ist, z.B. die Übersetzung in eine andere Sprache.

Eine erste Unterscheidung der vier verschiedenen Typen erhält man, wenn man sich die Länge der Zwischenresultate der generierten Wörter anschaut. Die Typ 1-Bedingung: die linke Seite einer Produktion ist immer kleiner oder gleich der rechten Seite, bewirkt, dass bei Typ 1,2 und 3 die Zwischenresultate der Produktionen immer nur länger oder gleich bleiben, nie kürzer werden. Das gilt nicht für Typ 0-Grammatiken. Dort kann die Länge der Zwischenresultate mal größer und mal wieder kleiner werden.

Die folgenden Bilder illustrieren den Unterschied:



Diese Beobachtung führt zu einem ersten signifikanten Ergebnis, welches einen Unterschied zwischen dem Typ 0 und den Typen 1,2,3 darstellt:

Theorem 1 (Entscheidbarkeit des Wortproblems für Typ 1,2,3) *Das Wortproblem für die Sprachen der Typen 1,2 und 3 ist entscheidbar.*

Das (zugegebenermaßen sehr ineffiziente) Entscheidungsverfahren funktioniert folgendermaßen: Angenommen das zu testende Wort w hat die Länge l_w. Wenn die Sprache vom Typ 1,2 oder 3 ist, gibt es auch eine Grammatik von diesem Typ. Ausgehend von der Startvariablen der Grammatik produziert man systematisch alle Wörter bis zur Länge l_w. Da die Produktionen die Länge der Wörter nicht verkleinern, hat man nach endlicher Zeit alle Wörter der Länge l_w produziert. Jetzt braucht man nur zu überprüfen, ob das Wort w dabei ist, oder nicht.

Dieses Verfahren funktioniert nicht, wenn die Sprache vom Typ 0 ist. Da die Produktionen die Wörter auch verkürzen können, kann man zu keinem Zeitpunkt entscheiden, ob man tatsächlich alle Wörter der Länge l_w produziert hat.

Zumindest bekommt man mit dieser Idee ein Semientscheidungsverfahren für Typ 0-Sprachen: Wenn das Wort w tatsächlich in der Sprache ist, dann wird es nach endlicher Zeit generiert werden. Falls nicht, gibt es kein Terminierungskriterium.

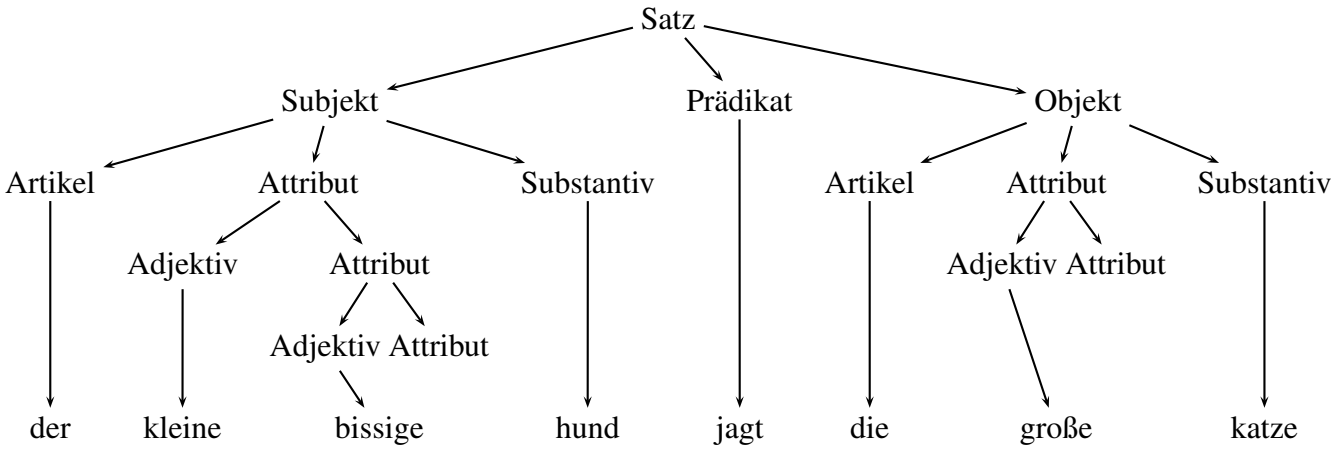
Es zeigt sich, dass das Wortproblem für Typ 0-Sprachen in der Tat nur semientscheidbar ist.

Für praktische Zwecke benutzt man natürlich nicht ein solches extrem ineffizientes Verfahren. Für jeden Typ der Sprache gibt es passendere viel effizientere Verfahren.

4 Syntaxbäume

Eine Folge von Regelanwendungen, ausgehend von der Startvariablen der Grammatik bis zum generierten Wort lässt sich für Typ 3- und Typ 2-Sprachen in einen *Syntaxbaum umwandeln*. Das illustrieren wir an dem natürlichsprachlichen Beispiel von oben: der kleine bissige hund jagt die große katze.

Der Syntaxbaum dazu sieht folgendermaßen aus.



Aus einem solchen Syntaxbaum kann man die einzelnen Komponenten, z.B. Subjekt, Prädikat, Objekt auslesen. Auch für Programme werden auf ähnliche Weise Syntaxbäume erzeugt, aus denen man die Bestandteile des Programms auslesen kann.

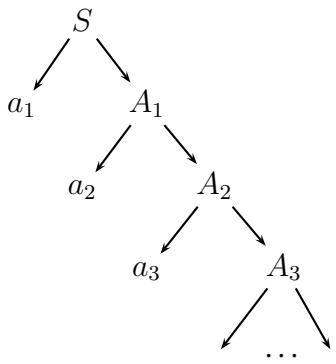
Theorem 2 (Konstruktion eines Syntaxbaumes)
Jeder Ableitung eines Wortes w in einer Grammatik vom Typ 2 oder 3 kann man einen Syntaxbaum zuordnen.

Die Konstruktion des Syntaxbaumes ist folgendermaßen:
Gegeben die Ableitung: $S \Rightarrow w_0 \Rightarrow w_1 \ldots \Rightarrow w$.
S wird der Wurzelknoten des Baums. Für jede Regelanwendung $L \to R$ bekommt der Knoten mit L für jedes Element von R einen Unterknoten.

Diese Konstruktion funktioniert nur für Typ 2- und Typ 3-Grammatiken, da nur diese auf der linken Seite der Regeln eine einzige Variable haben.

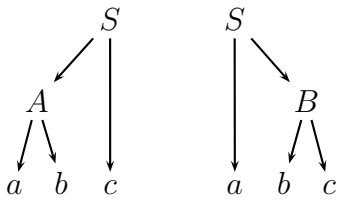
Allerdings sind die Syntaxbäume für Typ 3-Grammatiken zu einer linearen Kette degeneriert.

Bei Typ 3-Grammatiken sehen die Regelanwendungen ja so aus: $S \to a_1 A_1, A_1 \to a_2 A_2, \ldots$ Daraus ergibt sich ein Syntaxbaum zu:



Es wäre äußerst wünschenswert, wenn der Syntaxbaum für jedes Wort *eindeutig* wäre. Leider ist das nicht immer der Fall, wie folgendes Beispiel zeigt:

Die Produktionsregeln seien:
$S \to aB, S \to Ac, A \to ab, B \to bc.$
Für das Wort abc gibt es dann folgende zwei Syntaxbäume:



Solche uneindeutigen Grammatiken sind in der Informatik nicht sehr nützlich und sollten daher vermieden werden.

Man kennt dieses Phänomen auch aus der natürlichen Sprache. Im Satz „Ich sah den Mann mit dem Fernrohr“ ist es unklar, ob ich das Fernrohr habe und den Mann beobachte, oder ob der Mann das Fernrohr trägt. Solche Uneindeutigkeiten kann man meist erst aus dem Kontext auflösen.

5 Resultate

In diesem Kapitel werden einige Resultate aus den Untersuchungen über formale Sprachen überblicksmäßig und ohne Beweise zusammengefasst. Beweise oder Beweisideen findet man teilweise in den speziellen Miniskripten zu den einzelnen Sprachtypen, oder in der Literatur.

5.1 Das Wortproblem

Für Informatiker ist die Komplexität eines Problems von besonderem Interesse. Für das Wortproblem hat man die Komplexität der Algorithmen untersucht, die typischerweise für dessen Lösung benutzt werden. Dabei wird die Komplexität in der Anzahl von Rechenschritten, abhängig von der Größe der Eingabe, angegeben.

Folgendes ergibt sich für das Wortproblem, wobei n die Länge des Wortes ist.:

Typ 3	linear
Det. kf	linear
Typ 2	$\mathcal{O}(n^3)$
Typ 1	exponentiell (NP-hart)
Typ 0	unlösbar

Dabei steht Det. kf für die Menge der Typ 2-Sprachen, deren Wörter sich mit einem deterministischen Verfahren parsen lassen.

Die einzelnen Algorithmen werden in den dazugehörigen Miniskripten vorgestellt.

5.2 Weitere Probleme

Neben dem Wortproblem hat man noch weitere Probleme zu Formalen Sprachen untersucht:

Leerheitsproblem: ist die Sprache leer oder nicht leer? Nicht immer ist das einfach zu sehen.

Äquivalenzproblem: sind zwei Sprachen gleich oder nicht? Zwei verschiedene Grammatiken, sogar unterschiedlichen Typs können die gleiche Menge von Wörtern erzeugen.

Schnittproblem: ist der Schnitt von zwei Sprachen leer oder nicht?

Folgende Resultate über die Entscheidbarkeit dieser Probleme wurden bewiesen:

	Wortproblem	Leerheitsproblem	Äquivalenzproblem	Schnittproblem
Typ 3	ja	ja	ja	ja
Det. kf	ja	ja	ja	nein
Typ 2	ja	ja	nein	nein
Typ 1	ja	nein	nein	nein
Typ 0	nein	nein	nein	nein

5.3 Abschlusseigenschaften

Man sagt „eine Sprachklasse ist abgeschlossen unter der Operation X“, falls die Anwendung der Operation X auf eine oder mehrere Sprachen eines Typs wiederum eine Sprache dieses Typs ergibt. Als

Operationen kommen insbesondere die Mengenoperationen in Frage, aber auch die Konkatenation von Wörtern.

Folgende Operationen wurden untersucht:

Schnitt:	ist der mengentheoretische Schnitt $L_1 \cap L_2$ der zwei Sprachen.
Vereinigung:	ist die mengentheoretische Vereinigung $L_1 \cup L_2$ der zwei Sprachen.
Komplement:	ist das mengentheoretische Komplement L' der Sprache.
Produkt:	ist die Konkatenation $L_1 L_2$ von Wörtern der beiden Sprachen.
Stern:	ist das beliebige (incl. 0 mal) Aneinanderhängen L^* der Wörter der Sprache.

Die folgende Tabelle gibt die Resultate über die Abschlusseigenschaften wider.

	Schnitt	Vereinigung	Komplement	Produkt	Stern
Typ 3	ja	ja	ja	ja	ja
Det. kf	nein	nein	ja	nein	nein
Typ 2	nein	ja	nein	ja	ja
Typ 1	ja	ja	ja	ja	ja
Typ 0	ja	ja	nein	ja	ja

Typ 3, Schnitt, ja bedeutet z.B., dass der Schnitt zweier Typ 3-Sprachen wieder eine Typ 3-Sprache ist.

Interessant ist Typ 0, Komplement, nein. Es bedeutet, dass das Komplement einer Typ 0-Sprache nicht mehr mit einer Grammatik der Chomsky Hierarchie beschrieben werden kann. Eine Typ 0-Sprache ist z.B. die Menge der terminierenden Java-Programme. Für diese gibt es eine Typ 0-Grammatik. Für deren Komplement, nämliche die Menge der nichtterminierenden Java Programm, gibt es dagegen keine solche Grammatik mehr.

6 Miniskripten

In folgenden Miniskripten werden weitere Details zu formalen Sprachen besprochen:

TuringMaschinen:	führt das Konzept der Turingmaschinen ein,
Typ3Sprachen:	führt Typ 3-Grammatiken und endliche Automaten ein,
RegulaereAusdrueck:	führt reguläre Ausdrücke als spezielle Spezifikationssprache für Typ 3-Sprachen ein,
Typ2Sprachen:	führt Typ 2-Sprachen ein,
Kellerautomaten:	führt Kellerautomaten und Top-Down-Parsing für Typ 2-Sprachen ein,
BottomUpParsing:	erklärt Bottom-Up-Parsing und den CYK-Algorithmus für Typ 2-Sprachen,
ANTLR:	gibt eine Einführung in den Parsergenerator ANTLR, ein Java Programm,
Typ01:	bespricht die Sprachen vom Typ 1 und Typ 0.

Die Miniskripten sollten auch in dieser Reihenfolge gelesen werden.

Stichwortverzeichnis