Formale Sprachen vom Typ 1 und 0

Hans Jürgen Ohlbach

Keywords: Typ 1- und Typ 0-Sprachen, Syntaxgraphen, Korrespondenz zu Turingmaschinen, Nichtdeterminismus

Empfohlene Vorkenntnisse: Formale Sprachen, Typ 3- und Typ 2-Sprachen, Turingmaschinen

Inhaltsverzeichnis

1 Einführung

Bei formalen Sprachen vom Typ 0 und 1 dürfen, im Gegensatz zu Typ 2 und 3, die linken Seiten der Gramamtikregeln beliebig lang sein. Formale Sprachen vom Typ 0 haben Grammatiken, die keinen Einschränkungen genügen. Die Sprachen vom Typ 1, dagegen, haben die Einschränkung, dass die Produktionsregeln der Grammatik nie Wörter verkürzen. Die linken Seiten einer Produktion sind also kleiner oder gleich der rechten Seite.

Die folgende Grammatik illustriert den Unterschied der verschiedenen Produktionsregeltypen. Groß geschriebene Worte sind Variablen (Nicht-Terminalsymbole), klein geschriebene sind Konstanten (Terminalsymbole).

Outfit	$\rightarrow$	Hose Hemd Jacke	Typ 2
Hose	$\rightarrow$	langeHose | kurzeHose	Typ 3
Hemd	$\rightarrow$	t-shirt | langärmelig | kurzärmelig	Typ 3
Jacke	$\rightarrow$	jackett | strickjacke	Typ 3
langärmelig jackett	$\rightarrow$	langärmelig jackett krawatte	Typ 1
kurzeHose Hemd Jacke	$\rightarrow$	kurzeHose Hemd	Typ 0

Wenn man also ein langärmeliges Hemd und ein Jackett an hat, dann darf man auch noch eine Krawatte dazu anziehen. Wenn man allerdings eine kurze Hose, ein Hemd und eine Jacke an hat, dann kann man die Jacke auch ausziehen.

2 Beispiele

2.1 Typ 1

Wir betrachten zunächst ein Typ 1-Beispiel und illustrieren damit auch gleichzeitig, dass ein Top-Down-Verfahren zum Parsen von Wörten, ähnlich wie bei Typ 2-Sprachen nicht praktikabel ist.

Die folgende Typ 1-Grammatik erzeugt die Sprache $L = \{a^n b^n c^n | n > 0\}$.

1.	S	$\rightarrow$	aSBC	2.	S	$\rightarrow$	aBC				
3.	CB	$\rightarrow$	BC	4.	aB	$\rightarrow$	ab				
5.	bB	$\rightarrow$	bb	6.	bC	$\rightarrow$	bc	7.	cC	$\rightarrow$	cc.

Eine Ableitung des Wortes aaabbbccc sieht so aus:

S $\overset{2\times 1.}{\rightarrow}$ aaSBCBC $\overset{2.}{\rightarrow}$ aaaBCBCBC $\overset{3\times 3.}{\rightarrow}$ aaaBBBCCC $\overset{4.}{\rightarrow}$ aaabBBCCC
$\overset{2\times 5.}{\rightarrow}$ aaabbbCCC $\overset{6.}{\rightarrow}$ aaabbbcCC $\overset{2\times 7.}{\rightarrow}$ aaabbbccc.

Dieses Beispiel illustriert sehr deutlich, dass leider die Top-Down-Methode des Parsens, die bei Typ 2-Sprachen so erfolgreich ist, bei Typ 1 nicht mehr funktioniert.

Die Top-Down-Methode startet mit der Startvariablen im Stack, und expandiert immer die oberste

Variable im Stack. Sobald Konstanten auftauchen, werden sie mit dem Eingabewort verglichen, und gelöscht, wenn sie gleich sind.

Wir versuchen das mit diesem Beispiel, und sehen dann, wo es schief geht.

Wort	Stack	anzuwendende Regel
aaabbbccc	S	Regel 1
aaabbbccc	aSBC	löschen
aabbbccc	SBC	Regel 1
aabbbccc	aSBCBC	löschen
abbbccc	SBCBC	Regel 2
abbbccc	aBCBCBC	löschen
bbbccc	BCBCBC	

Jetzt müsste Regel 4 angewendet werden, aber das passende a ist gelöscht. Wenn man stackartig weitermachen wollte, ist keine Regel mehr anwendbar. Man könnte jetzt weiter unten im Stack linke Seiten von Regeln finden und dort weitermachen.

Wort	Stack	anzuwendende Regel
bbbccc	BCBCBC	Regel 3
bbbccc	BBCCBC	Regel 3
bbbccc	BBCBCC	Regel 3
bbbccc	BBBCCC	

Jetzt ist die Ableitung leider endgültig in einer Sackgasse.

Der Unterschied zu Typ 2 ist: Die Typ 2-Produktionen sind *kontextfrei*, d.h. auf der linken Seite der Produktionsregeln steht nur eine einzige Variable. Hat man ein Wort, das aus Variablen und Konstanten gemischt ist, kann man die Variablen in *beliebiger* Reihenfolge expandieren (durch rechte Seiten ersetzen). Im Top-Down-Verfahren geht man da von links nach rechts vor.

Typ 1- und Typ 0-Grammatiken sind kontextsensitiv. D.h. auf der linken Seite können mehr als ein Symbol stehen. Bei einem Wort, das aus Variablen und Konstanten gemischt ist, kann man da nicht mehr in beliebiger Reihenfolge expandieren. Die Anwendbarkeit einer Regel kann von der Anwendung anderer Regeln abhängig sein. Auch eine Vorausschau ist da kaum möglich.

Natürlich kann man, zumindest bei Typ 1-Sprachen eine vollständige Suche mit Backtracking machen. Aber ohne Vorausschau ist das extrem aufwendig.

Die Top-Down-Verfahren sind daher aus Effizienzgründen nicht praktikabel.

2.2 Typ 0

Die folgende Grammatik erzeugt die Sprache $L = \{a^{2^n} \mid n \geq 0\}$.

1.	S	$\rightarrow$	ACaB	Beginn der Ableitung, erzeugt schon das erste a
2.	Ca	$\rightarrow$	aaC	verdoppelt die a's in einem Lauf von links nach rechts
3.	CB	$\rightarrow$	DB,	wandelt den „Verdoppler" C in den „Rückläufer" D
4.	CB	$\rightarrow$	E,	beginnt am Ende, wenn alle a's erzeugt sind, das „Aufräumen"
5.	aD	$\rightarrow$	Da,	D läuft von rechts nach links über alle a's hinweg, und wird ...
6.	AD	$\rightarrow$	AC,	... mit dieser Regel wieder (von A) als neuer „Verdoppler" eingesetzt
7.	aE	$\rightarrow$	Ea,	sorgt für den Lauf der „Aufräum-Variablen" E von recht nach links
8.	AE	$\rightarrow$	ϵ	eliminiert die letzten Variablen und belässt das gewünschte a-Wort

Regel 4 und 8 sind Typ 0-Regeln.

Eine Ableitung des Wortes aaaaaaaa (= a^{2^3}) sieht so aus:

S	$\overset{1.}{\rightarrow}$ ACaB	$\overset{2.}{\rightarrow}$ AaaCB	a einmal verdoppelt
	$\overset{3.}{\rightarrow}$ AaaDB	$\overset{2\times5.}{\rightarrow}$ ADaaB	D nach links (zweimal)
	$\overset{6.}{\rightarrow}$ ACaaB	$\overset{2\times2.}{\rightarrow}$ AaaaaCB	a zweimal verdoppelt
	$\overset{3.}{\rightarrow}$ AaaaaDB	$\overset{4\times5.}{\rightarrow}$ ADaaaaB	D nach links (viermal)
	$\overset{6.}{\rightarrow}$ ACaaaaB	$\overset{4\times2.}{\rightarrow}$ AaaaaaaaCB	a viermal verdoppelt
	$\overset{4.}{\rightarrow}$ AaaaaaaaE	$\overset{8\times7.}{\rightarrow}$ AEaaaaaaaa	E nach links (achtmal)
	$\overset{8.}{\rightarrow}$ aaaaaaaa		

Auch an diesem Beispiel könnte man verdeutlichen, dass das Top-Down-Verfahren nicht so funktionieren kann, wie bei Typ 2-Sprachen.

3 Bottom-Up-Verfahren

Die Idee bei Bottom-Up-Verfahren ist: man startet mit dem zu parsenden Wort und sucht darin rechte Seiten von Produktionsregeln der Grammatik, die man dann durch die entsprechende linke Seite ersetzt. Das macht man so lange bis nur noch die Startvariable übrig bleibt.

Beispiel: Wir benutzen die Grammatik für die Sprache $\{a^n b^n c^n \mid n > 0\}$ (siehe oben).

Die Grammatik ist wie oben

1.	S	$\rightarrow$	aSBC	2.	S	$\rightarrow$	aBC					
3.	CB	$\rightarrow$	BC	4.	aB	$\rightarrow$	ab					
5.	bB	$\rightarrow$	bb	6.	bC	$\rightarrow$	bc	7.	cC	$\rightarrow$	cc.

Das zu parsende Wort sei aaabbbccc. Eine erfolgreiche Sequenz von Ersetzungen ist:

Schritt	Wort	Regel
1	aaabbbccc	7
2	aaabbbccC	7
3	aaabbbcCC	6
4	aaabbbCCC	5
5	aaabbBCCC	5
6	aaabBBCCC	4
7	aaaBBBCCC	3

Schritt	Wort	Regel
8	aaaBBCBCC	3
9	aaaBBCCBC	3
10	aaaBCBCBC	2
11	aaSBCBC	1
12	aSBC	1
13	S	

Es gäbe aber noch viele andere Möglichkeiten, rechte Seiten durch linke Seiten zu ersetzen. Die allermeisten davon würden in eine Sackgasse führen. Eine Vorausschau, um die richte Regelanwendung zu finden ist auch kaum möglich. Daher ist dieses Verfahren hochgradig nichtdeterministisch, und für praktische Anwendungen ziemlich ungeeignet.

Das Beispiel zeigt ein Charakteristikum für Typ 1-Sprachen: die Anzahl von Symbolen in der Ersetzungssequenz wird nie größer. Sie bleibt entweder gleich, oder wird kleiner. Man sagt, der benötigte Speicher ist *linear beschränkt*. Das liegt daran, dass bei Typ 1-Sprachen die rechten Seiten der Regeln nie kürzer als die linken Seiten sind. Die Menge aller möglichen Ersetzungssequenzen ist daher endlich. Eine systematische Suche aller Ersetzungssequenzen terminiert deshalb.

Falls man das Bottom-Up-Verfahren mit einer Turingmaschine implementiert, reicht dazu eine *linear beschränkte Turingmaschine*. Sie braucht auf dem Band nicht mehr Platz, als das zu analysierende Wort.

Dies ist bei Typ 0-Sprachen nicht mehr der Fall. Hier können die Wörter zwischendurch auch wieder länger werden. Daher terminiert das Verfahren nicht mehr notwendigerweise.

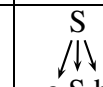
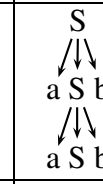
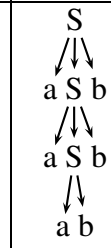
4 Syntaxbäume und Syntaxgraphen

Für die wenigsten Anwendungen eines Parsings reicht eine ja/nein-Entscheidung aus. Um ein erfolgreich geparstes Wort, z.B. ein Programmtext, weiterverarbeiten zu können, braucht man als Ergebnis des Parsens eine Datenstruktur, aus der man die einzelnen Komponenten des Worts/Programms zur Weiterverarbeitung entnehmen kann. Hierfür eignet sich sehr gut ein *Syntaxbaum*. Syntaxbäume lassen sich für Typ 2-Sprachen auf sehr natürliche Weise erzeugen. Leider klappt das bei Typ 1- und Typ 0-Sprachen nicht mehr. Was da schief geht, sieht man am Vergleich zwischen einer Typ 2-Sprache und einer Typ 1-Sprache.

Als Typ 2-Sprache betrachten wir $\{a^n b^n \mid n > 0\}$ mit der Grammatik

$$S \to aSb \mid ab.$$

Das Wort aaabbb lässt sich damit folgendermaßen ableiten:

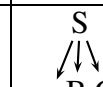
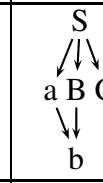
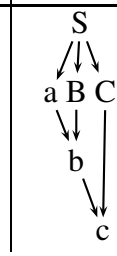
Ableitung	Syntaxbaum
S $\rightarrow$ aSb	
$\rightarrow$ aaSbb	
$\rightarrow$ aaabbb	

Links steht die Ableitung, und rechts, wie sich daraus der Syntaxbaum entwickelt.
Da aus der Grammatik immer nur eine Variable ersetzt wird, lässt sich der Syntaxbaum systematisch aufbauen, indem man unter dem Knoten für die zu ersetzende Variable den rechten Teil der Produktionsregel hängt.

Jetzt versuchen wir etwas analoges für die schon bekannte Typ 1-Sprache $\{a^n b^n c^n \mid n > 0\}$ mit der auch schon bekannten Grammatik

1.	S	$\rightarrow$	aSBC	2.	S	$\rightarrow$	aBC					
3.	CB	$\rightarrow$	BC	4.	aB	$\rightarrow$	ab					
5.	bB	$\rightarrow$	bb	6.	bC	$\rightarrow$	bc	7.	cC	$\rightarrow$	cc.

Das einfachste Wort, dass man mit dieser Grammatik ableiten kann, ist abc.

Ableitung	Syntaxgraph
S $\rightarrow$ aBC	
$\rightarrow$ abC	
$\rightarrow$ abc	

Da bei Typ 1-Grammatiken mehr als ein Symbol auf der linken Seite der Produktionsregeln stehen darf, kann es keinen *Syntaxbaum* mehr geben, sondern eher einen *Syntaxgraphen*. Inwieweit das eine brauchbare Datenstruktur für eine Weiterverarbeitung ist, ist unklar.

5 Sprachen und Turingmaschinen

Wie wir gesehen haben, lässt sich das Wortproblem für Typ 1- und Typ 0-Sprachen mit Bottom-Up-Parsing lösen. Diese Methode kann man auch mit Turingmaschinen implementieren, wobei man bei Typ 1-Sprachen sogar mit linear beschränkten Turingmaschinen auskommt.

Für jede solche Sprache gibt es also eine Turingmaschine, die das Wortproblem dadurch löst, dass man das Wort auf das Band schreibt, und die Turingmaschine rechte Seiten von Produktionsregeln durch linke Seiten ersetzen lässt. Terminiert sie in einem Endzustand, dann ist das Wort akzeptiert, ansonsten nicht.

Viel interessanter ist die umgekehrte Richtung: gegeben eine Turingmaschine, gibt es dann auch eine entsprechende Grammatik, die genau diejenigen Wörter erzeugt, bei der die Turingmaschine mit dem Wort als Eingabe im Endzustand terminiert?

Um es etwas präziser zu sagen definieren wir die *Sprache einer Turingmaschine*:

Definition 1 (Sprache einer Turingmaschine) *Für eine Turingmaschine M besteht die Sprache $L(M)$ genau aus den Eingaben auf dem Band, bei der die Turingmaschine in einem Endzustand terminiert. (Dabei ist es unerheblich, was zum Schluss auf dem Band steht).*

Es ist in der Tat möglich, eine Turingmaschine M in eine Grammatik G zu übersetzen, die genau die Wörter $L(M)$ produziert. Ist die Turingmaschine linear beschränkt, dann erhält man eine Typ 1-Grammatik, ansonsten eine Typ 0-Grammatik.

Wir illustrieren die Übersetzung zunächst an einem einfachen Beispiel. Unsere Turingmaschine zählt die Anzahl von 1en in einer Bitfolge, und terminiert wenn es eine gerade Anzahl ist. Ihre Sprache ist also die Menge der Bitfolgen mit einer geraden Anzahl von 1en. 10111 gehört also dazu, aber 1011 nicht.

Wir brauchen drei Zustände z_g (für gerade), z_u (für ungerade) und z_e als Endzustand. z_g ist der Startzustand.

Die Übergänge sind:

$$
\begin{array}{lcll}
z_g, 0 & \mapsto & z_g, 0, R & \text{Die Maschine geht also einmal durch die Bitfolge. Wenn sie eine 1 trifft,} \\
z_g, 1 & \mapsto & z_u, 1, R & \text{schaltet sie zwischen den Zuständen } z_g \text{ und } z_u \text{ um. Sobald sie das Blank} \\
z_g, \_ & \mapsto & z_e, \_, N & \text{trifft, und ist im Zustand } z_g, \text{ dann kommt sie in den Endzustand. Ist sie} \\
z_u, 0 & \mapsto & z_u, 0, R & \text{jedoch im Zustand } z_u, \text{ dann geht es nicht mehr weiter.} \\
z_u, 1 & \mapsto & z_g, 1, R &
\end{array}
$$

Daraus möchten wir jetzt eine Grammatik machen. Diese soll genau die Bitfolgen mit einer geraden Anzahl von 1en erzeugen.

Man geht in zwei Schritten vor:

1. Erzeuge eine beliebige Bitfolge, mit zusätzlichen Steuervariablen.
2. Simuliere die Arbeit der Turingmaschine, so dass am Ende nur für die richtigen Bitfolgen ein Wort der Sprache entsteht.

Für den ersten Teil nehmen wir folgende Grammatikregeln:

$S \to A1' \mid A0'$
$A \to A1 \mid A0 \mid (z_g, 0) \mid (z_g, 1).$

Diese Regeln erzeugen Wörter wie $(z_g, 0)11010'$ oder $(z_g, 1)1101'$ usw.

Die Konstanten $0'$ und $1'$ stehen für 0 bzw. 1, markieren aber die letzten Position.

$(z_g, 0)$ und $(z_g, 1)$ haben den Status von *Variablen*, nur etwas komplizierter geschrieben. Es ist z_g weil z_g der Startzustand ist.

Für einen Zustand z und ein Bit b simuliert eine Sequenz $0110(z, b)011$, dass die Maschine gerade im Zustand z ist, und der Schreib-Lesekopf über dem b steht. Der Bandinhalt selbst ist also $0110b011$.

$(z_g, 0)11010'$ bedeutet also: Bandinhalt 011010 (mit Blanks davor und dahinter), Zustand z_g, und der Schreib-Lesekopf steht über dem ersten Bit.

Die anschließenden Produktionen der Grammatik müssen jetzt die Arbeitsweise der Turingmaschine simulieren.

Turingmaschine	Grammatik		
$z_g, 0 \mapsto z_g, 0, R$	$(z_g, 0)0$	$\to$	$0(z_g, 0)$
	$(z_g, 0)1$	$\to$	$0(z_g, 1)$
	$(z_g, 0')$	$\to$	0
$z_g, 1 \mapsto z_u, 1, R$	$(z_g, 1)0$	$\to$	$1(z_u, 0)$
	$(z_g, 1)1$	$\to$	$1(z_u, 1)$
$z_g, \_ \mapsto z_e, \_, N$			
$z_u, 0 \mapsto z_u, 0, R$	$(z_u, 0)0$	$\to$	$0(z_u, 0)$
	$(z_u, 0)1$	$\to$	$0(z_u, 1)$
$z_u, 1 \mapsto z_g, 1, R$	$(z_u, 1)0$	$\to$	$1(z_g, 0)$
	$(z_u, 1)1$	$\to$	$1(z_g, 1)$
	$(z_u, 1')$	$\to$	1

Wir illustrieren die Produktion der Grammatik mit aus den Startregeln erzeugbarem Wort:
$(z_g, 0)101' \mapsto 0(z_g, 1)01' \mapsto 01(z_u, 0)1' \mapsto 010(z_u, 1') \to 0101.$

Ein Wort mit ungerader Anzahl 1en würde mit einer Variablen enden, die nicht mehr ersetzt werden kann:
$(z_g, 0)10' \mapsto 0(z_g, 1)0' \mapsto 01(z_u, 0').$

Das Beispiel ist insofern ein Spezialfall, als die Turingmaschine an dem Eingabestring nichts verändert. Im allgemeinen Fall, wo sie die Eingabe beliebig manipulieren kann, muss man die Kodierung in die

Grammatik noch etwas erweitern. Anstelle der Wörter, die die Startregeln erzeugen, wo das Zielwort schon direkt da steht, verdoppelt man die Buchstaben zunächst, z.B.
$(z_g, (0,0))(1,1)(1,1)(0,0)(1,1)(0', 0')$.
Jetzt hat man in jedem Paar einen Buchstaben, der verändert werden darf, sowie eine Originalkopie, die bis zum Schluss erhalten bleibt. Dafür braucht man noch eine Nachbearbeitung, die die Paare auf die Originalkopie reduziert.

Mit diesem Trick kann man jede beliebige Turingmaschine in eine Grammatik umbauen, die genau die Sprache der Turingmaschine erzeugt.

Was ist nun der Unterschied zwischen allgemeiner Turingmaschine und linear beschränkter Turingmaschine? Die Turingmaschine im obigen Beispiel ist linear beschränkt, und die daraus erzeugte Grammatik hat Typ 1-Regeln.

Ist die Turingmaschine nicht linear beschränkt, dann vergrößert sie den Bandinhalt während der Rechnung. Für die Grammatik bedeutet das, dass das Endergebnis, was ja zu Beginn von den Startregeln erzeugt wird, kleiner ist als die Zwischenergebnisse. Um am Ende wieder auf das kürzere Endergebnis zu kommen, braucht man verkürzende Regeln, und die sind nicht mehr Typ 1, sondern Typ 0.

Damit können wir zusammenfassen:

Theorem 1 *Für jede Typ 0-Grammatik G gibt es eine Turingmaschine M mit $L(G) = L(M)$, und umgekehrt.*

Für jede Typ 1-Grammatik G gibt es eine linear beschränkte Turingmaschine M mit $L(G) = L(M)$, und umgekehrt (Satz von Kuroda).

6 Determinismus und Nichtdeterminismus

Das Bottom-Up-Parsing ist zunächst nichtdeterministisch, sowohl für Typ 0 als auch für Typ 1. Um ein vollständiges Verfahren zu bekommen, muss man also systematisch alle Alternativen ausprobieren. Implementiert man es in einer allgemeinen Turingmaschine, dann ist es kein Problem, sich die noch nicht getesteten Alternativen auf dem Band zu merken. Das Band ist ja beliebig groß. Man kann daher *systematisch*, d.h. *deterministisch* alle Alternativen durchprobieren.

Es gibt daher hier keinen Unterschied zwischen deterministischen und nichtdeterministischen Typ 0-Sprachen.

Bei Typ 1-Sprachen ist man auf den beschränkten Platz der linear beschränkten Turingmaschine angewiesen. Vermutlich reicht der Platz dann nicht, aus, sich die noch nicht getesteten Alternativen zu merken. Es ist jedoch bisher noch niemand gelungen, zu beweisen, dass man dem linearen Platz nicht auskommt, oder ein anderes Verfahren zu finden, welchen mit dem Platz auskommt. *Es ist noch ein ungelöstes Problem (das erste LBA-Problem).*

Daher ist es bisher unklar, ob es einen Unterschied zwischen deterministischen und nichtdeterministischen Typ 1-Sprachen gibt.