Top-Down-Parsing / Kellerautomaten

Hans Jürgen Ohlbach

Keywords: Typ 2-Sprachen, Top-Down-Parsing, Kellerautomaten

Empfohlene Vorkenntnisse: Formale Sprachen, Typ 2-Sprachen

Inhaltsverzeichnis

1 Einführung

Die Typ 2-Sprache $L = \{a^n b^n \mod n > 0\}$ macht deutlich, dass zum Parsen von Wörtern wie $aaabbb$ ein *Gedächtnis* nötig ist. Man muss sich die Anzahl a's merken, um es mit der Anzahl b's vergleichen zu können. In den Typ 2-Grammatiken erkennt man das auch daran, dass auf den rechten Seiten der Produktionen dieselben Symbole mehrfach, oder auch nur dieselbe Anzahl Symbole vorkommen können. Die Grammatik für die Sprache L hat die Produktionen $S \to ab \mid aSb$ woraus ersichtlich ist, dass gleich viele as und bs vorkommen müssen. Das kann man in den zu testenden Wörtern nur erkennen, wenn man sich etwas merken kann.

Die endlichen Automaten für reguläre Sprachen haben zunächst kein Gedächtnis. Man könnte sie aber mit einem Gedächtnis ausstatten, damit sie sich etwas merken können. Die endlichen Automaten hatten den Vorteil, dass man mit ihnen ein einziges mal durch das Wort läuft, und dann ist das Parsen des Wortes fertig. Um diesen Vorteil zu behalten, und auch, um keine teuren Operationen auf einem Gedächtnis einzuführen, bietet sich als einfachste Form eines Gedächtnisses ein Stack (Kellerspeicher) an. Bei jedem Schritt des Automaten kann man dann etwas in das Stack schreiben (push) oder etwas aus dem Stack holen (pop). Das verteuert das Parsing nur minimal.

Es zeigt sich, dass die Typ 2-Sprachen genau die Sprachen sind, deren Wörter mit einem endlichen Automaten, angereichert durch ein Stack, geparst werden können. Das sind die *Kellerautomaten* (engl. Push-Down Automaton, PDA).

2 Kellerautomaten

Kellerautomaten haben gegenüber den endlichen Automaten folgende Änderungen

- sie haben zusätzlich ein Stack, das bei jedem Übergang gelesen oder beschrieben werden kann.
- in das Stack kann man auch Zeichen schreiben, die nicht im Alphabet der Sprache sind
- sie akzeptieren ein Wort, wenn *das Wort und der Keller* leer geworden sind. Endzustände sind damit unnötig. Um das leer gewordene Stack vom Startzustand unterscheiden zu können, enthält es i.A. zu Beginn ein spezielles Zeichen.

Definition 1 (Kellerautomat (engl. Pushdown Automaton, PDA))
Ein Kellerautomat $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ *besteht aus*

- *einer Zustandsmenge* Z*,*
- *dem Alphabet der Sprache* Σ*,*
- *dem Kelleralphabet* Γ*, das sind die Zeichen, die in den Keller geschrieben werden können,*
- *der Überführungsfunktion* $\delta : Z \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Z \times \Gamma^*)$*,*
- *dem Startzustand* z_0 *und*
- *dem untersten Kellerzeichen* $\# \in \Gamma$*.*

Die Überführungsfunktion δ ist folgendermaßen zu verstehen: Sie testet den momentanen Zustand Z, den ersten Buchstaben des zu untersuchenden Wortes, und das oberste Kellerelement (top of stack). Abhängig von diesen Werten, geht sie in einen neuen Zustand über und ersetzt das oberste Kellerelement durch kein, ein, oder mehrere neue Elemente. δ kann nichtdeterministisch sein, d.h. für den gleichen Zustand, und für den gleichen Buchstaben kann es verschiedene Folgezustände geben (daher das Potenzmengenzeichen $\mathcal{P}$ bei δ).

Beispiel: Wir definieren eine Kellerautomaten zum Parsen der Sprache $\{a^n b^n \mid n > 0\}$. Der Automat pusht für jedes gelesene a ein A auf das Stack, bis er auf ein b trifft. Für jedes b poppt er ein Element vom Stack. Wir starten das Stack mit dem Zeichen #.

Man kann das mit folgenden Regeln aufschreiben, die die Übergangsfunktion definieren:

1. $z_0, a, \# \mapsto z_0, A$
2. $z_0, a, A \mapsto z_0, AA$
3. $z_0, b, A \mapsto z_1, \epsilon$
4. $z_1, b, A \mapsto z_1, \epsilon$

Die Regeln bedeuten dabei:
Zustand vorher, 1. Zeichen im Wort, top-of-stack $\rightarrow$ Zustand nachher, Ersatz für top-of-stack.
ϵ entspricht der pop-Funktion (Der Erstatz für top-of-stack ist leer).

Die erste Regel besagt: wenn im Zustand z_0 ein a gelesen wird, dann bleibe im Zustand z_0 und ersetze das Startsymbol # durch A.

Die zweite Regel besagt: wenn im Zustand z_0 ein a gelesen wird, und das top-of-stack A ist, dann bleibe im Zustand z_0 und ersetze das A durch AA. Das wäre die übliche push-Operation.

Die dritte Regel greift wenn das erste b gelesen wird. Dann wird das A vom Stack „gepoppt“, und er geht in den Zustand z_1.

Die vierte Regel liest ein b und poppt ein A von Stack.

Der Automat stoppt mit „akzeptiert“ wenn das Wort und das Stack leer geworden sind.

Beispiel: Für das Wort $aabb$ würde sich die *Konfiguration* (Zustand, Restwort, Stack) des Automaten folgendermaßen entwickeln:

Zustand	Wort	Stack
z_0	$aabb$	$\#$
z_0	abb	A
z_0	bb	AA
z_1	b	A
z_1	ϵ	ϵ

2.1 Grammatik $\mapsto$ Kellerautomat

Der Automat aus dem Beispiel oben ist ein handcodierter Kellerautomat, genau für diese Sprache programmiert. Was man jedoch haben möchte ist ein *automatisiertes* Verfahren, für eine Typ 2-Sprache einen Kellerautomaten zu erzeugen, der als Parser fungiert.

Wie das gehen könnte illustrieren wir an derselben Sprache. Allerdings nehmen wir dazu die Grammatik als Ausgangspunkt.

Die Produktionsregeln sind. $S \rightarrow ab \mid aSb$

Der Kellerautomat startet mit der Startvariablen S im Stack (anstelle von #). Die Idee ist jetzt, Variablen im top-of-stack, die als linke Seiten einer Produktion vorkommen, im top-of-stack durch ihre rechten Seiten zu ersetzen. Erscheinen dabei Konstante im top-of-stack, dann kann man sie mit den Buchstaben im Wort vergleichen. Sind beide identisch, werden sie gelöscht. Sind beide verschieden, muss der Automat backtracken.

Die folgenden Regeln für die Übergangsfunktion realisieren das.

1. $z, \epsilon, S \quad \mapsto \quad z, ab$
2. $z, \epsilon, S \quad \mapsto \quad z, aSb$
3. $z, a, a \quad \mapsto \quad z, \epsilon$
4. $z, b, b \quad \mapsto \quad z, \epsilon$

Regel 1 ist aus der Produktion $S \to ab$ erzeugt. Sie besagt: wenn das top-of-stack S ist, dann ersetze S durch ab.

Regel 2 ist aus der Produktion $S \to aSb$ erzeugt. Sie besagt: wenn das top-of-stack S ist, dann ersetze S durch aSb.

Regel 3 und 4 vergleichen das Wort mit dem top-of-stack. Bei Übereinstimmung löschen sie die Buchstaben aus dem Wort und dem Stack.

Wir verdeutlichen die Arbeitsweise am Wort $aabb$. Die Konfiguration des Automaten besteht aus dem Zustand, dem Wort und dem Stackinhalt, zu Beginn: $z, aabb, S$

Der Automat führt folgende Schritte aus:

1. Da das top-of-stack S ist, sind Regel 1 und 2 anwendbar. Bei Anwendung von Regel 1 läuft der Automat in eine Sackgasse, und muss backtracken. Die richtige Regel ist Regel 2. Es ergibt sich die Abfolge:

1.	$z, aabb, S$	Regel 2:	z, ϵ, S	$\mapsto$	z, aSb
2.	$z, aabb, aSb$	Regel 3:	z, a, a	$\mapsto$	z, ϵ
3.	z, abb, Sb	Regel 1:	z, ϵ, S	$\mapsto$	z, ab
4.	z, abb, abb	Regel 3:	z, a, a	$\mapsto$	z, ϵ
5.	z, bb, bb	Regel 4:	z, a, a	$\mapsto$	z, ϵ
6.	z, b, b	Regel 4:	z, a, a	$\mapsto$	z, ϵ
7.	z, ϵ, ϵ				

Wort und Stack sind leer geworden. Daher ist das Wort akzeptiert.

Das Beispiel illustriert mehrere Dinge:

- es ist nur ein einziger Zustand notwendig. Den kann man auch weglassen.
- die Arbeitsweise kann nichtdeterministisch werden. Dann muss der Automat evtl. backtracken. (Bei praktische relevanten Sprachen sollte man das weitgehend vermeiden. Mit einer gewissen Vorausschau können die Automaten aber oft die richtige Entscheidung treffen.)
- Die Übergangsfunktion lässt sich aus der Grammatik automatisch erzeugen:
  Eine Produktion $X \to Y_1 \dots Y_k$ wird übersetzt in. $z, \epsilon, X \mapsto z, Y_1 \dots Y_k$.
  Für jede Konstante a braucht man eine Regel $z, a, a \mapsto z, \epsilon$.

Wenn man die Übersetzung einer kontextfreien Grammatik in einen Kellerautomaten exakt definiert, dann kann man folgendes Theorem beweisen. Der Beweis ist technisch, braucht aber keine tieferen Einsichten.

Theorem 1 *Eine Sprache ist genau dann kontextfrei wenn ihr Wortproblem mit einem Kellerautomaten gelöst werden kann.*

Beispiel: Als weiteres Beispiel betrachten wir ein kleine arithmetische Sprache:

$$S \to 0 \mid \ldots \mid 9 \mid (S+S) \mid (S*S)$$

Sie beschreibt Ausdrücke wie $(3+(5*6))$.

Da nur ein Zustand gebraucht wird, lassen wir ihn ganz weg.

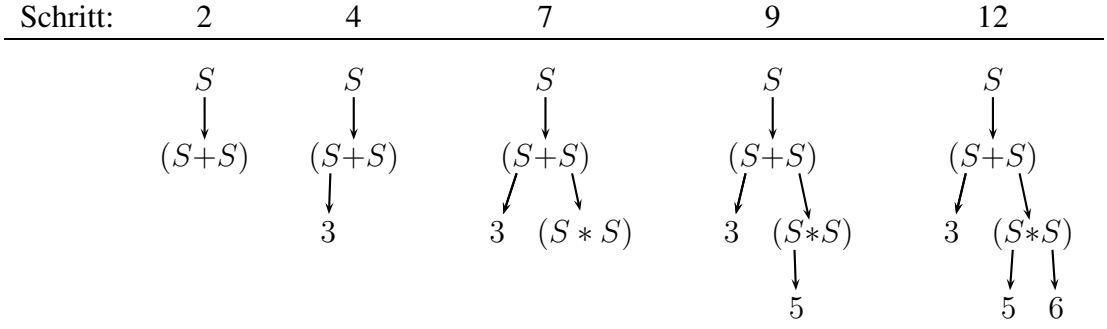
Die Übergangsfunktion ergibt sich jetzt aus den Regeln

1.	ϵ, S	$\mapsto$	0	13.	$0, 0$	$\mapsto$	ϵ	25.	$+, +$	$\mapsto$	ϵ
...	...	$\mapsto$		...	...	$\mapsto$		26.	$*, *$	$\mapsto$	ϵ
10.	ϵ, S	$\mapsto$	9	22.	$9, 9$	$\mapsto$	ϵ				
11.	ϵ, S	$\mapsto$	$(S+S)$	23.	$(, ($	$\mapsto$	ϵ				
12.	ϵ, S	$\mapsto$	$(S*S)$	24.	$),)$	$\mapsto$	ϵ				

Die Regeln 1-12 sind extrem nichtdeterministisch. Mit etwas Vorausschau lassen sich aber die richtigen Übergänge auswählen. Beim Parsen von $(3+(5*6))$ ergibt sich jetzt folgende Abfolge:

1	$(3+(5*6)), S$	Regel 11:	ϵ, S	$(S+S)$	8	$5*6)), S*S))$	Regel 6:	ϵ, S	5
2	$(3+(5*6)), (S+S)$	Regel 23:	$(, ($	ϵ	9	$5*6)), 5*S))$	Regel 18:	$5, 5$	ϵ
3	$3+(5*6)), S+S)$	Regel 4:	ϵ, S	3	10	$*6)), *S))$	Regel 26:	$*, *$	ϵ
4	$3+(5*6)), 3+S)$	Regel 16:	$3, 3$	ϵ	11	$6)), S))$	Regel 7:	ϵ, S	6
5	$+(5*6)), +S)$	Regel 25:	$+, +$	ϵ	12	$6)), 6))$	Regel 19:	$6, 6$	ϵ
6	$(5*6)), S)$	Regel 12:	ϵ, S	$(S*S)$	13	$)),))$	Regel 24:	$),)$	ϵ
7	$(5*6)), (S*S))$	Regel 23:	$(, ($	ϵ	14	$),)$	Regel 24:	$),)$	ϵ
					15	ϵ, ϵ			

Verfolgt man die Schritte, die linke Seite einer Produktion durch rechte Seite ersetzen, kann man sofort einen Syntaxbaum erzeugen.



Diesen Syntaxbaum könnte man dann weiter nutzen, um z.B. den arithemetischen Ausdruck auszurechen. Beim Parsen von Programmen entsteht auch ein Syntaxbaum, der dann der Ausgangspunkt für eine Übersetzung in Maschinensprache ist.

Top-Down-Parsen mit Kellerautomaten sieht zwar auf den ersten Blick hochgradig nichtdeterminis-

tisch, und damit ineffizient aus. Aber durch begrenzte Vorausschau können die Parser meistens die richtige Entscheidung treffen. Indem man die Grammatiken entsprechend formuliert, kann man den Parsern auch entscheidend helfen.

Grammatiken für Programmiersprachen enthalten Regeln wie

Anweisung	$\rightarrow$	if-Anweisung | while-Anweisung | Zuweisung
if-Anweisung	$\rightarrow$	if(Bedingung) then {Anweisung} else {Anweisung}
while-Anweisung	$\rightarrow$	while(Bedingung) {Anweisung}
Zuweisung	$\rightarrow$	Variable := Term
Bedingung	$\rightarrow$	...
Term	$\rightarrow$	...

die alle deterministisch sind.

Leider ist es aber, anders als bei regulären Sprachen, *nicht* möglich, die Kellerautomaten *immer* deterministisch zu machen.

Oft kombiniert man auch kontextfreie Grammatiken mit regulären Ausdrücken, so dass man eventuellen Nichtdeterminismus in den regulären Ausdrücken in *deterministische* Automaten übersetzen kann, die man in den Parsvorgang einbaut.

3 Parsergeneratoren

Wie wir gesehen haben lassen sich kontextfreie Grammatiken vollautomatisch in Kellerautomaten umwandeln. Dafür hat man sog. *Parsergeneratoren* entwickelt. Sie lesen eine Grammatik ein, und erzeugen einen Parser. Diesen Parser kann man auf ein konkretes Wort anwenden, und erhält einen Syntaxbaum, oder eine Fehlermeldung. Für den Syntaxbaum gibt es dann einen Schnittstelle, mit der man ihn von einer Programmiersprache ansprechen und weiterverarbeiten kann.

Im C und C++-Umfeld gibt es als Parsergeneratoren Flex und Bison. Für Java gibt es ANTLR. Beide sind professionelle Werkzeuge für realistische Anwendungen.