

Teil 3 Bottom-Up-Parser, Kontextfreie Sprachen

An einem früheren Beispiel demonstrieren wir eine Analyse-Methodik, die völlig anders vorgeht als die Top-Down-Analyse des letzten Teils:

- Statt (wie in Teil 2, **Top-Down**) im Stack ein abstraktes Abbild des aktuellen *Reststrings* zu verwalten und durch systematisches *Expandieren* von Hilfssymbolen das abstrakte Stack-Abbild und den Reststring aneinander anzupassen,
- werden nun (**Bottom-Up**) solange Zeichen des Inputstrings in den Stack eingelesen, bis dort die Rechte Seite einer Grammatik-Regel komplett vorliegt, welche dann zur linken Seite der Regel (also zu einem Hilfssymbol) *reduziert* wird.

Aktueller Stackinhalt und Reststring ergänzen sich jeweils zu einem kompletten Abbild des Inputstrings: Der Stackinhalt ist ein abstraktes Bild des schon gelesenen Inputs, der Reststring hingegen ist unverändert.

Man beachte: In diesem Teil ist das **top-Element** des Stacks grundsätzlich **rechts** notiert.

Beispiel (arithmetische Ausdrücke)

Wir wählen eine vereinfachte Form der Grammatik:

Grammatik:

A ->	T+A	T
T ->	F*T	F
F ->	(A)	a

Beispielstring:

a	+	a	*	a
1	2	3	4	5

Stack	Reststring	Aktion
€	a + a * a	SHIFT
a	+ a * a	REDUCE mit F->a
F	+ a * a	REDUCE mit T->F
T	+ a * a	SHIFT
T +	a * a	SHIFT
T + a	* a	REDUCE mit F->a
T + F	* a	SHIFT
T + F *	a	SHIFT
T + F * a	€	REDUCE mit F->a
T + F * F	€	REDUCE mit T->F
T + F * T	€	REDUCE mit T->F*T
T + T	€	REDUCE mit A->T
T + A	€	REDUCE mit A->T+A
A	€	Stop, String akzeptiert

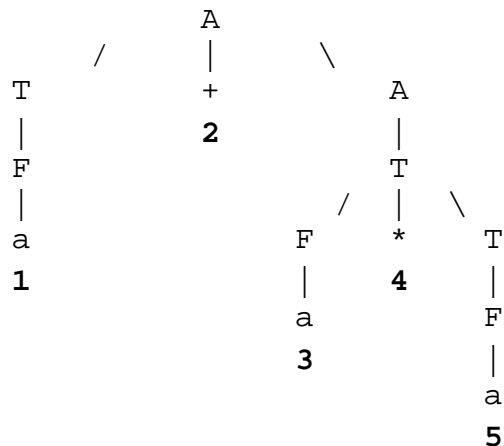
Wenn wir die Reduktionen von unten nach oben zusammensetzen, dann erhalten wir die zugehörige **Rechtsableitung** (*rechts*, weil stets das am weitesten *rechts* stehende Zeichen ersetzt wird):

```

A -> T + A
  -> T + T
  -> T + F * T
  -> T + F * F
  -> T + F * a
  -> T + a * a
  -> F + a * a
  -> a + a * a

```

Wir notieren auch den **Syntaxbaum**, der bei dieser Methodik von unten nach oben (**Bottom-Up**) entstanden ist:



Es ist evident, dass die obige Analyse eine Anzahl von Willkürlichkeiten enthält: Zweimal wurde *a* über *F* nach *T* reduziert, einmal jedoch nur bis *F*. Einmal wurde *T* nach *A* reduziert, einmal beließen wir es bei *T*.

Diese Willkürlichkeiten resultierten natürlich aus Vorwissen über das gewünschte Ziel, und unserer Aufgabe ist es, diese Einsichten dem System beizubringen.

Um die Argumentation so einfach wie möglich zu halten, werden wir in Folgenden stets ohne Lookahead arbeiten (innerhalb **LR(0)** in der späteren Notation), wir haben also die Vorstellung, daß der Restinput völlig verborgen ist.

Prüft man unter diesem Aspekt das obige Beispiel, so erkennt man: Ohne Lookahead bestünden manchmal Zweifel über die erforderliche Reduktion, mit Lookahead hätte man stets Eindeutigkeit.

Es ist einleuchtend, daß im Vergleich zu den Top-Down-Parsern von Teil 2 die Bottom-Up-Parser grundsätzlich mächtiger sein sollten: Während bei Bottom-Up-Reduktion die komplette rechte

Seite der Ersetzungsregel vom Parser bereits gesehen wurde, müssen LL(1)-Parser die richtige Ersetzungsregel finden, von deren rechter Seite sie nur das erste Zeichen kennen.

Da die erforderlichen Konstruktionen recht komplex werden, wählen wir ein sehr einfaches Beispiel (entnommen aus Asteroth/Baier: Theoretische Informatik).

Beispiel (Ausdrücke der Form $a^n b^n$, $n \geq 1$)

Grammatik: $S \rightarrow C$
 $C \rightarrow aCb \mid ab$

Beispielstring: a a b b
 1 2 3 4

Stack	Reststring	Aktion
ε	a a b b	SHIFT
a	a b b	SHIFT
a a	b b	SHIFT
a a b	b	REDUCE mit $C \rightarrow ab$
a C	b	SHIFT
a C b	ε	REDUCE mit $C \rightarrow aCb$
C	ε	REDUCE mit $S \rightarrow C$
S	ε	Stop, String akzeptiert

Die nötigen Hilfestellungen für die fälligen Entscheidungen (ist SHIFT oder REDUCE aktuell nötig? falls REDUCE, dann mit welcher Regel?) bekommt man wiederum nur durch systematische Analyse der Grammatik. Wir notieren eine Auswahl dieser Entscheidungshilfen:

Stack- inhalt v für v gültige Items		

I_0	ε	$S \rightarrow \bullet C$, $C \rightarrow \bullet aCb$, $C \rightarrow \bullet ab$
I_1	a, aa	$C \rightarrow \bullet ab$, $C \rightarrow \bullet aCb$, $C \rightarrow a \bullet b$, $C \rightarrow a \bullet Cb$
I_2	aC	$C \rightarrow aC \bullet b$
I_3	ab, aab	$C \rightarrow ab \bullet$
I_4	aCb	$C \rightarrow aCb \bullet$
I_5	C	$S \rightarrow C \bullet$

Diese Regeln sind folgendermaßen zu interpretieren, z.B. Zeile I_1 :
 Bei Stackinhalt a oder aa sind überhaupt nur die vier rechts

- daneben notierten Produktionen aussichtsreich, und zwar würde
- in den Fällen $C \rightarrow a \bullet b$, $C \rightarrow a \bullet Cb$ das oberste auf dem Stack liegende Zeichen a bei der Reduktion benutzt.
 - in den anderen beiden Fällen $C \rightarrow \bullet ab$, $C \rightarrow \bullet aCb$ würde dagegen vom aktuellen Stackinhalt gar nichts benutzt, die potentielle Reduktion würde vielmehr komplett den noch nicht gescannten, verborgenen Bereich des Inputs betreffen.

Diese Grundidee notieren wir etwas formaler:

Gültige Items

Eine **punktierte Produktion** $A \rightarrow y_1 \bullet y_2$ kann zum Zuge kommen in einer Situation, wo der vordere Teil y_1 oben auf dem Stack liegt und der hintere Teil y_2 dem Anfang des noch verborgenen Inputs entspricht.

Eine punktierte Produktion $A \rightarrow y_1 \bullet y_2$ ist **gültig für** den Stackinhalt $v := xy_1$, wenn bei "Vorgeschichte" v und geeigneter Realisierung des verborgenen Inputs die Reduktion $A \rightarrow y_1 y_2$ zu einer kompletten Rechtsableitung führt.

Itemlisten können für die Bottom-Up-Analyse sehr nützlich sein:

Stack	Reststring	Aktion
...		
a a	b b	SHIFT (1)
a a b	b	REDUCE mit $C \rightarrow ab$ (2)
...		

	v	für v gültige Items
I_1	a, aa	$C \rightarrow \bullet ab$, $C \rightarrow \bullet aCb$, $C \rightarrow a \bullet b$, $C \rightarrow a \bullet Cb$
I_3	ab, aab	$C \rightarrow ab \bullet$
...		

In der Situation (2) zeigt ein Blick auf die Itemliste (Zeile I_3), daß die komplette Rechte Seite einer Produktion schon auf dem Stack liegt und daß man reduzieren kann.

In der Situation (1) zeigt Zeile I_1 der Itemliste für jeden der vier in Frage kommenden Fälle, daß noch keine komplette Rechte Seite einer Produktion auf dem Stack liegt und daß man erst noch shiften sollte.

Diese Argumentation hat aber noch erhebliche Mängel:

Die Effizienz ist problematisch, denn gültige Items gibt es viele, und Stackinhalte ("Vorgeschichten") gibt es wie Sand am Meer.

Außerdem (das vor allem) ist zu klären, ob der Ansatz methodisch überhaupt gut genug ist. Denn es könnte ja sein, daß in einer

konkreten Situation sowohl Reduzieren als auch Shiften möglich ist, daß aber nur eine dieser beiden Möglichkeiten zum Erfolg (zu einer kompletten Rechtsableitung) führt. – Wir werden sehen, wie diese Probleme sich der Reihe nach klären.

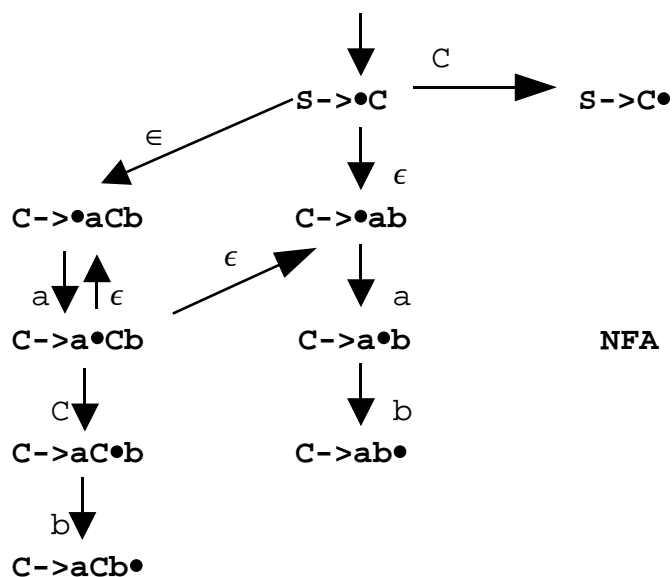
Die folgende rekursive Charakterisierung wird es uns ermöglichen, die gültigen Items effizient zu verwalten:

Gültige Items (rekursive Beschreibung)

Sei S das Startsymbol der Grammatik, A und B seien Hilfssymbole und a ein Terminalzeichen. Wende so oft wie möglich die folgenden Regeln an:

- (1) Alle Items der Form $S \rightarrow \bullet y$ sind gültig für den leeren String
- (2) Ist $A \rightarrow y_1 \bullet a y_2$ gültig für v , so ist $A \rightarrow y_1 a \bullet y_2$ gültig für va
 \Rightarrow Kante mit Bewertung a von $A \rightarrow y_1 \bullet a y_2$ nach $A \rightarrow y_1 a \bullet y_2$
- (3) Ist $A \rightarrow y_1 \bullet B y_2$ gültig für v , so ist $A \rightarrow y_1 B \bullet y_2$ gültig für vB
 \Rightarrow Kante mit Bewertung B von $A \rightarrow y_1 \bullet B y_2$ nach $A \rightarrow y_1 B \bullet y_2$
- (4) Ist $A \rightarrow y_1 \bullet B y_2$ gültig für v , dann sind alle Items der Form $B \rightarrow \bullet y$ gültig für v .
 \Rightarrow unbewertete Kante $A \rightarrow y_1 \bullet B y_2$ nach $B \rightarrow \bullet y$

Wie schon angedeutet, läßt sich das Ergebnis dieser rekursiven Beschreibung grafisch veranschaulichen:



Anhand dieses Graphen läßt sich die Entwicklung der gültigen Items im Verlauf der Analyse gut beschreiben, etwa so:

"Ist für irgendein v das Item $C \rightarrow \bullet ab$ gültig, so ist für va das Item $C \rightarrow a \bullet b$ gültig."

"Ist für v das Item $C \rightarrow a \bullet Cb$ gültig, so sind für das gleiche v die Items $C \rightarrow \bullet aCb$ und $C \rightarrow \bullet ab$ gültig."

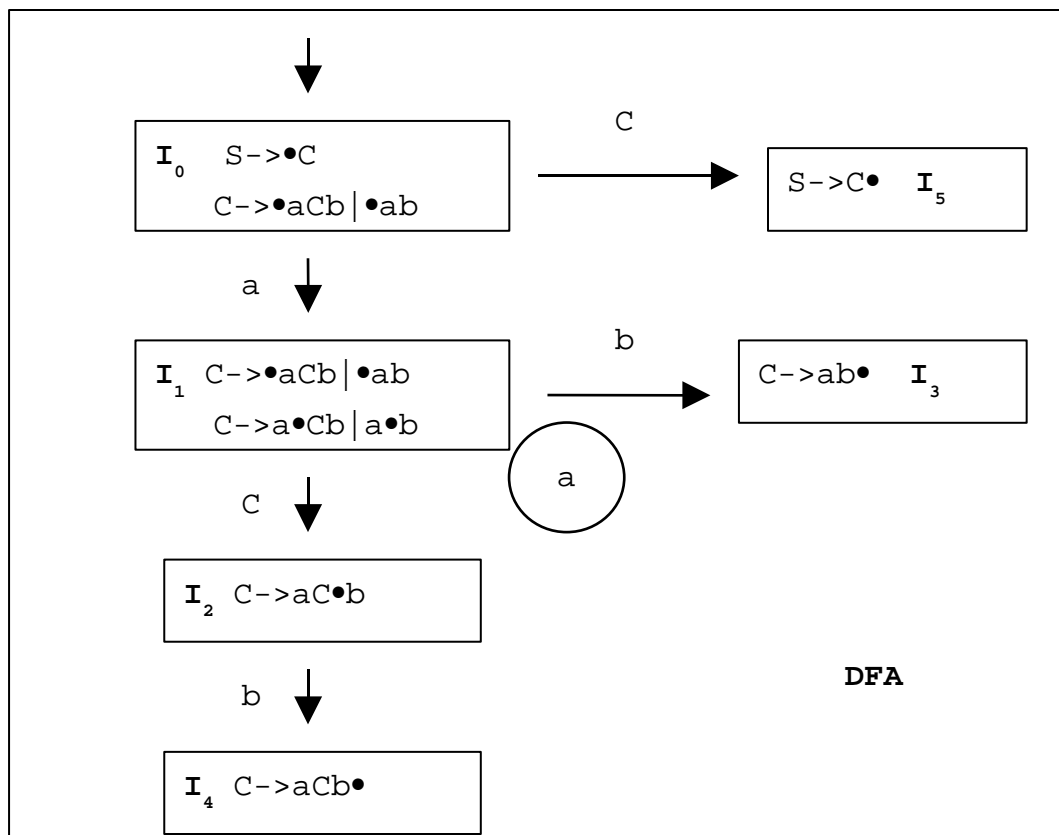
Übersetzt in eine andere, aus Teil 1 bekannte Sprechweise:

"Sind wir im Zustand $C \rightarrow \bullet ab$, so kommen wir mit a in den Zustand $C \rightarrow a \bullet b$ "

"Sind wir im Zustand $C \rightarrow a \bullet Cb$, so sind wir gleichzeitig auch in den Zuständen $C \rightarrow \bullet aCb$ und $C \rightarrow \bullet ab$."

Man erkennt, daß sich die Fortentwicklung der gültigen Items über den obigen Graphen beschreiben läßt, sofern man den als nichtdeterministischen Automaten NFA interpretiert!

Praktischer jedoch ist der zugehörige *deterministische* Automat DFA, dessen "Superzustände" alle für einen Stackinhalt v gültigen Items zusammenfassen. Konstruiert wird dieser deterministische Automat mit dem aus Teil 1 bekannten Potenzmengenprozeß:



Dieser deterministische Automat ist nun die Maschinerie, die uns bei der Inputanalyse (zumindest bei diesem Beispiel) sicher den Weg weisen wird. - Wir wählen unseren alten Inputstring:

Beispielstring: a a b b
 1 2 3 4

Wie starten in Zustand I_0 .

Wir shiften a und kommen nach I_1 .

Wir shiften das zweite a und bleiben in I_1 .

Wir shiften b und kommen nach I_3 .

Im Zustand I_3 haben wir nun die komplette rechte Seite der Produktion $C \rightarrow ab\bullet$ auf dem Stack und können reduzieren, d.h. wir poppen b, poppen a und pushen C.

Für den Graph bedeutet das, dass wir die Kanten, über die wir nach I_3 kamen, rückwärts gehen, d.h. zurück nach I_1 und dann einmal den Loop bei I_1 entlang. Pushen von C bedeutet Übergang nach I_2 .

In I_2 shiften wir das zweite b, kommen nach I_4 und haben wieder die komplette rechte Seite einer Produktion $C \rightarrow aCb\bullet$ auf dem Stack und können reduzieren. Wir gehen also den entsprechenden Weg I_2, I_1, I_0 zurück, pushen C und landen in I_5 .

Hier können wir mittels $S \rightarrow C\bullet$ wieder reduzieren. Wir haben nun den Input aabb komplett gelesen und ihn auf das Startsymbol S reduziert.

Wir stoppen und erklären den Inputstring für akzeptiert.

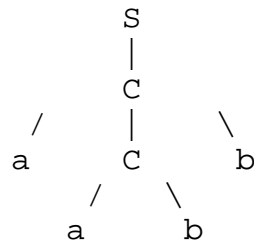
Man beachte: Beim Reduzieren brauchen wir die Informationen über die im Graphen zurückgelegten Wege. Aus diesem Grunde pushen wir beim Zustandsübergang nicht nur das Zeichen, sondern auch den neuen Zustand. - Hier der komplette Analyseverlauf:

Stack	Rest	Aktion
I_0	a a b b	SHIFT
I_0aI_1	a b b	SHIFT
$I_0aI_1aI_1$	b b	SHIFT
$I_0aI_1aI_1bI_3$	b	REDUCE mit $C \rightarrow ab\bullet$
$I_0aI_1CI_2$	b	SHIFT
$I_0aI_1CI_2bI_4$	ϵ	REDUCE mit $C \rightarrow aCb\bullet$
I_0CI_5	ϵ	REDUCE mit $S \rightarrow C\bullet$
I_0	ϵ	Stop, String akzeptiert

Wir notieren noch die resultierende **Rechtsableitung**:

$S \rightarrow C$
 $\rightarrow aCb$
 $\rightarrow a a b b$

und den Bottom-Up entstandenen **Syntaxbaum**:



Es ist üblich, die in dem obigen deterministischen Automaten DFA enthaltenen Informationen in **Parse-Tabellen** zu notieren.

Die **GOTO-Tabelle** codiert den Graphen, die **Action-Tabelle** gibt an, ob zu shiften oder zu reduzieren ist:

GOTO

	a	b	C
I ₀	I ₁		I ₅
I ₁	I ₁	I ₃	I ₂
I ₂		I ₄	
I ₃			
I ₄			
I ₅			

ACTION

	Action
I ₀	SHIFT
I ₁	SHIFT
I ₂	SHIFT
I ₃	REDUCE mit $C \rightarrow ab\bullet$
I ₄	REDUCE mit $C \rightarrow aCb\bullet$
I ₅	ACCEPT (REDUCE mit $S \rightarrow C\bullet$)

Man beachte: Offensichtlich wird die Bottom-Up-Analyse nicht immer so gut funktionieren wie bei diesem "gutartig" gewählten Beispiel. Zwei Probleme sind vorstellbar:

REDUCE-REDUCE-Konflikt: Ein Zustand des DFA enthält zwei unterschiedliche Items der Form $A \rightarrow y\bullet$. Dann wüßte man nicht, mit welcher Regel man reduzieren soll.

SHIFT-REDUCE-Konflikt: Ein Zustand enthält ein Item der Form $A \rightarrow y\bullet$ und ein anderes der Form $A \rightarrow y_1\bullet y_2$. Dann wüßte man nicht, ob man reduzieren oder shiften soll.

Wir machen es uns einfach und sagen: Eine Grammatik ohne REDUCE-REDUCE-Konflikte und ohne SHIFT-REDUCE-Konflikte ist eine **LR(0)-Grammatik** (von Links her scannen, Rechtsableitung erzeugen, kein Lookahead)

Bei all unseren Beispielen der Teile 2 und 3 handelte es sich um nichtreguläre formale Sprachen, die wir Top-Down oder Bottom-Up und mit Stack-Unterstützung erfolgreich geparkt haben. Zumindest aus theoretischer Sicht ist es an der Zeit zu klären, innerhalb welcher Sprachklasse wir uns hier eigentlich bewegt haben.

Prinzipiell neu war nur der Stack, den wir zum Speichern von Teilresultaten benötigten. Aus der Sicht der Automatentheorie hatten wir es immer mit *Kellerautomaten* zu tun:

Definition (deterministischer Kellerautomat)

Zu einem deterministischen Kellerautomaten $A=(Q,\Sigma,q_0,p,F)$ gehört

- eine endliche Menge Q von Zuständen mit Startzustand q_0 und Menge F von Ja-Zuständen.
- das Alphabet Σ
- die Zustandsüberföhrungsfunktion (das *Programm*)

$$p: Q \times \Sigma \times \Sigma \dashrightarrow Q \times \{N,R\} \times \Sigma^*$$

Die Funktion p ist so zu verstehen, daß in Abhängigkeit vom aktuellen Zustand, vom aktuellen Eingabezeichen und vom obersten Kellerzeichen

- über den neuen Zustand entschieden wird,
- entschieden wird, ob der Lesekopf sich nach rechts bewegt oder ob er stehen bleibt
- das oberste Kellerzeichen durch einen (evtl. leeren) String ersetzt wird.

[Letzteres eine Verallgemeinerung der beiden Standardaktionen pop bzw. push, bei denen das oberste Kellerzeichen gepoppt bzw. ein *einzelnes* Zeichen gepusht wird.]

Der Kellerautomat **akzeptiert** einen Eingabestring, wenn er sich nach Einlesen des kompletten Strings in einem Ja-Zustand befindet und wenn dann außerdem der Keller leer ist.

Die von einem Kellerautomaten **akzeptierte Sprache** ist die Menge aller akzeptierten Strings.

Die regulären Sprachen hatten wir außer mittels DFA/NFA-Automaten auf äquivalente Weise auch über reguläre Grammatiken und reguläre Ausdrücke definieren können. Naheliegend ist daher die Frage, ob bei den aktuellen nichtregulären Sprachen etwas ähnliches möglich ist, ob wir also (wenn wir uns auf die

Grammatiken konzentrieren) einen Grammatiktyp finden können, der in seiner Ausdruckstärke äquivalent zu Kellerautomaten ist. Zu diesem Zweck prüfen wir die Grammatiken unserer Beispiele genauer. Typisch waren wie im letzten Beispiel Grammatikregeln der Art

$$C \rightarrow aCb$$

die bei regulären Grammatiken nicht zugelassen waren, die aber immer noch die grundsätzliche Eigenschaft haben, daß auf der linken Seite der Produktion genau ein Hilfssymbol steht. Etwas Komplizierteres, also Grammatikregeln wie etwa

$$CB \rightarrow BC$$

$$0B \rightarrow 01 \quad (\text{Teil 1, Aufgabe 4})$$

wäre (für Parserzwecke jedenfalls) sicherlich sehr unpraktisch.

Aus grammatikalischer Sicht hatten wir es bei unseren nicht-regulären Beispielen stets mit *kontextfreie Grammatiken* zu tun:

Definition (kontextfreie Grammatik)

Eine **Grammatik** ist **kontextfrei**, wenn ihre Ersetzungsregeln alle von der Form

$$V \rightarrow str$$

sind, wobei V für ein Hilfssymbol steht und str für einen String.

Eine formale **Sprache** ist **kontextfrei**, wenn es zu ihr eine kontextfreie Grammatik gibt.

Mit dem Namen *kontextfrei* hat es Folgendes auf sich: Die Regel

$$V \rightarrow str$$

besagt, daß V überall lokal durch str ersetzt werden darf.

Hingegen ist es bei der Regel

$$uVw \rightarrow u str w \quad (u, w \text{ Strings})$$

lediglich erlaubt, bei Vorfinden des Verbundes uVw (also nur *im Kontext* uVw) die Variable V durch str zu ersetzen.

Es bleibt nun zu klären, ob unsere beiden neuen Begriffe, *Kellerautomat* und *kontextfreie Grammatik*, zueinander äquivalent sind, ob sie also ein-und-dieselbe Sprachklasse definieren. Man kann beweisen, dass jede durch einen deterministischen Kellerautomaten akzeptierte formale Sprache auch durch eine kontextfreie Grammatik abgeleitet werden kann. Die Umkehrung gilt allgemein aber nicht:

Beispiel

Wir betrachten die "ungeraden" binären Palindrome, also Palindrome mit zentralem Bit:

1 0 0 1 0 0 1 , 1 0 0 0 1

[Die Beschränkung auf die *ungeraden* Palindrome Bitzahl dient lediglich zur Vereinfachung der folgenden Argumentationen.]

Es ist einleuchtend, daß diese Sprache durch deterministische Kellerautomaten nicht erkannt werden kann. Denn der Automat (auch wenn er einen Keller hat) "weiß" nicht, wann die Mitte des Strings erreicht ist. Formal beweist man die Tatsache mit Hilfe einer verallgemeinerten Version des Pumping-Lemmas.

Eine kontextfreie Grammatik für die ungeraden Palindrome anzugeben ist hingegen fast trivial:

$S \rightarrow 0 \mid 1 \mid 0S0 \mid 1S1$

Wenn nun die simplen Palindrome so "kompliziert" sind, daß sie mit Kellerautomaten nicht geparkt werden können, so stellt sich die Frage "wie denn?". Später in Teil 3 werden wir die noch mächtigeren Turingmaschinen vorstellen, aber hier wollen wir zunächst den relativ rechenaufwendigen **CYK-Algorithmus** (Cocke/Younger/Kasami) vorführen, der bei allen kontextfreien Sprachen funktioniert und der dort in der Lage ist, die Wörter der betreffenden Sprache zu erkennen.

Zunächst (im Beispiel der ungeraden Palindrome) formen wir die obige Grammatik in zwei Schritten um:

$S \rightarrow 0 \mid 1 \mid ASA \mid BSB$
 $A \rightarrow 0$
 $B \rightarrow 1$

und dann

$S \rightarrow 0 \mid 1 \mid AU \mid BV$
 $U \rightarrow SA$
 $V \rightarrow SB$
 $A \rightarrow 0$
 $B \rightarrow 1$

Chomsky-Normalform

Das unten notierte Rechentableau zeigt, wie der CYK-Algorithmus das Beispiel-Palindrom

0 1 1 0 1 0 1 1 0

mittels **Dynamischer Programmierung** erkennt:

0	1	1	0	1	0	1	1	0
SA	SB	SB	SA	SB	SA	SB	SB	SA
V	V	U	V	U	V	V	U	
		S	S	S				
		U	V	V				
		S						
		V						
	S							
	U							
S								

[Prinzip erläutern]

Ob oder ob nicht der gegebene binäre String ein ungerades Palindrom ist, erkennt man nach Ausfüllen des Tableaus daran, ob das untere linke Kästchen das Startsymbol S enthält (warum?).

Würde man den Algorithmus programmieren, so wären offensichtlich drei Schleifen nötig. Daraus erkennt man die Größenordnung der benötigten Rechenzeit:

Satz: Der Rechenaufwand zum Ausfüllen des Tableau beim CYK-Algorithmus ist $O(n^3)$, wenn n Länge des Inputstrings ist.

Um die Effizienz des CYK-Algorithmus beurteilen zu können, muß man mit den früheren deterministischen Automaten und Kellerautomaten vergleichen. Ein deterministischer Automat bewegt bei jedem Maschinentakt den Lesekopf eine Position vorwärts. Die Rechenzeit ist also proportional zur Länge n des Inputstrings, d.h. $O(n)$ in der üblichen Notation.

Ähnliches gilt für Kellerautomaten, obgleich bei denen der Lesekopf manchmal (nämlich bei Expansion oder Reduktion einer Produktion der Grammatik) in Ruhe verharret.

Generell ist der CYK-Algorithmus mit wesentlich mehr Aufwand verbunden als der Betrieb von Automaten und Kellerautomaten.

Übungen zu TI, Teil 3

(1) Mit dem Bottom-Up-Parser der Vorlesung analysiere man die Strings aaabbb, aabbb und aaabb

(2a) Man bestimme den Bottom-Up-Parser (insbesondere den deterministischen Automaten DFA) für die Grammatik

$$S \rightarrow Ab, A \rightarrow Aa \mid a$$

und analysiere einige Inputstrings.

(b) Man probiere das Gleiche für die Grammatik

$$S \rightarrow Ab, A \rightarrow aA \mid a$$

(3) Man rechne den CYK-Algorithmus (für Palindrome ungerader Bitzahl) an den Beispielen

1 0 1 0 1 und 1 1 0 1 0